Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

FB Elektrotechnik und Informationstechnik

Security Vulnerabilities and Mitigation Strategies for Time-Triggered Real-Time Systems

Kristin Krüger

Dissertation 2026

Security Vulnerabilities and Mitigation Strategies for Time-Triggered Real-Time Systems

Vom Fachbereich Elektrotechnik und Informationstechnik
der Rheinland-Pfälzischen Technischen Universität Kaiserslautern-Landau
zur Verleihung des akademischen Grades

Doktor der Ingenieurswissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Kristin Krüger

geboren in Schwetzingen, Deutschland

D 386

Dekan	:	Prof. Dr.-Ing. Daniel Görges
Vorsitzender der Prüfungskommission	:	Prof. Dr.-Ing. Wolfgang Kunz
Gutachter	:	Prof. Dipl.-Ing. Dr. Gerhard Fohler
		Prof. Dr. Sebastian Altmeyer

Tag der Einreichung	:	17.07.2026
Tag der Verteidigung	:	14.09.2026

Kristin Krüger: *Security Vulnerabilities and Mitigation Strategies for Time-Triggered Real-Time Systems*, © 15.09.2026

ERKLÄRUNG

Ich versichere, dass ich diese Dissertation selbst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die aus den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Diese Dissertation wurde weder als Ganzes noch in Teilen als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht.

Es wurde weder diese noch eine andere Abhandlung bei einem anderen Fachbereich oder einer anderen Universität als Dissertation eingereicht.

Kaiserslautern, 15.09.2026

Kristin Krüger

千里の道も一歩から

Even a long journey begins with a single step.

ACKNOWLEDGMENTS

Someone once claimed that these pages are the last pages a PhD candidate writes and, even when the technical part of the thesis is complete, somehow these pages are the hardest to write. I think they are right on both accounts. People like to read their name on these pages, alongside kind words, and you want to give proper credit to the people that supported you, reciprocate their kindness and extend your gratitude. However, even with your best intentions, I think a certain discrepancy always remains. It takes skill to adequately put your feelings into words, so here is my best shot at it.

Gerhard, thank you for your unwavering support, the excellent guidance and compassion you have shown, and helping me to develop so many soft and hard skills. You inspired me with your leadership and your ability to have interesting discussions about literally anything. Also, Loriot became more popular thanks to you.

Marcus, thank you for all the technical discussions, co-authoring so many of my papers, and the very nice BBQ invitations. You offered support I never took for granted and much more than you had to. I'm glad you showed so much interest in my research.

Sebastian, I'm glad you joined me on this journey. Thank you for taking the time to read and review my thesis.

Thanks to all my former and current colleagues: Steffi, Markus, Peter, Rodrigo, Ali, Ankit, Carlos, Alexandre, Florian, Javier, Gabriele, Marine, Gautam, Luiz, Ibrahim, Isser, Shreya, and Allan. Also thanks to all my co-authors over the years that I haven't already mentioned: Nils, Richard, Martina, Paulo, Rafal, Sebastian, Federico, and Azin. Thanks for the technical discussions, brainstorming, and proofreading. Also thanks for pranking me by hiding my celebratory bottles; complaining how I could leave five adults alone on their way home through dark woods; pranking the whole chair by placing googly eyes everywhere; sharing tears and laughter as well as recipes; making sense of all these forms; nerding about our hobbies and showing off our newest technical gadgets; sharing traveling and life stories; helping each other through hard times, German grammar, and German bureaucracy (basically the same thing); infesting the chair with card game afternoons; supporting each other during COVID; making walking through the corridor feel like crossing military airspace because of tiny but vicious quadcopters; confiding in me when you felt uneasy or needed advice; designing the PCB that made everything come together; scribbling inadequate graffiti on the online whiteboard; betting on who would die and how during the last two seasons of Game of Thrones; making these BBQs so spontaneous; daring to ask why there was a medieval sword in my office; not noticing the cat on the ceiling for a full year; helping me with the airline that did not want to take my money; all the birthday parties and paper celebrations; planting vegetables and flowers in the raised bed;

sending me cheerful messages during sick leave; cherishing my Christmas Glögg; keeping the computers, servers, and robots alive; sharing pictures of memes, cats, and your newest creations with me; enjoying the afternoon sun for the last time this year, no, really for the last time this time after this day there will be no more sun Kristin stop laughing you must take this matter seriously; talking walks through the forest together; asking me questions about my native language I never anticipated anyone would ask; making me paint more 3D models for PhD hats than I wish to count; taking it with grace that I made the whole chair drink green tea at least once; and I could go on. Thanks for showing me that hard work, compassion and fun can go hand in hand.

My former Hiwis, Christian, Angelina, Bhakti, Saroja, Jonathan, Fatos, and Tobias, thank you for your hard work and dedication. Special thanks go out to Varun and especially Shreya who helped me implementing and evaluating my mitigation strategies.

Christian, Lena, Florian, Thomas, Merle, Simon, Jojo, and Karina, I am glad to have you in my life. You celebrated good times and were there in bad times. I appreciate all the support you have shown me. You did not stop believing in me and I am grateful for that. All your kind gestures and words of encouragement helped immensely.

Lastly, I want to thank my family, especially my grandparents. Even when you're no longer here, this is for you. Thank you for believing in me.

Now I'm off to another adventure. I hope I'll be able to tell as many tales about it as I can about this one.

Kaiserslautern, 15.09.2026
Kristin Krüger

P.S.: In dieser Dissertation verstecken sich zwei Katzen.

ABSTRACT

The number of security incidents in recent years shows a considerable increase in cyber attacks. The targets of these attacks are diverse, but also increasingly safety-critical systems like airplanes or power plants fall victim to attackers. These attacks show that even embedded systems cannot be considered inaccessible to malevolent actors anymore. As a consequence, security has become a primary concern for embedded systems and thus also for real-time systems which are often deployed as embedded systems in a safety-critical environment. Real-time systems are not only constrained by limitations typical for embedded systems, like power or memory, but they must also fulfill timing constraints. Thus, any security measure employed on a real-time system must adhere to timing constraints. However, these constraints allow for new attack vectors targeting the timing of the system. For example, a single deadline miss in a hard real-time system can already threaten system safety and have catastrophic consequences for the system or its environment. This inherent vulnerability of real-time systems to attacks in the time domain poses new and real-time specific challenges to security.

This thesis analyses threats to and vulnerabilities of real-time systems with a focus on time-triggered real-time systems. It shows how attackers may threaten fundamental system assumptions. The fault model of a system usually considers failures to be independent from each other, but an attacker violates this independence assumption when provoking failures. Moreover, an attacker may cause the system to reach a state that was not considered when worst case execution times were derived. As a consequence, these worst case execution time bounds may prove to be unsafe in a system under attack. Another vulnerability this thesis presents lies within a desirable system property itself. Time-triggered real-time systems provide deterministic scheduling behaviour. Deterministic behaviour is desirable for safety-critical systems as it allows for easy derivation of safe bounds on worst case execution time and simplifies system certification. However, the predictable schedules of time-triggered systems also ease attacks which exploit timing information for precisely timed attacks.

In this thesis, we detail these threats, present attack vectors for single core as well as multicore time-triggered real-time systems and introduce mitigation strategies tailored to these systems. The mitigation strategies employ randomization to prevent precisely timed attacks while also guaranteeing that all tasks meet their deadlines. We implement our mitigation strategies on FreeRTOS and demonstrate their low runtime overheads and small memory requirements.

TABLE OF CONTENTS

1 Introduction 1
1.1 Problem Statement . . . 2
1.2 Contribution of Thesis . . . 3
1.3 Outline . . . 4
Publications & Funding 7
2 Background 9
2.1 Real-Time Systems . . . 9
2.1.1 Tasks . . . 10
2.1.2 Timing Constraints . . . 11
2.1.3 Real-Time Scheduling . . . 12
2.1.4 Real-Time Operating Systems . . . 14
2.2 Fault-Tolerant and Dependable Systems . . . 16
2.2.1 Fault, Error, Failure . . . 17
2.2.2 Fault Handling and Fault Tolerance . . . 19
2.3 Time-Triggered Systems . . . 20
2.4 Security . . . 21
2.5 Related Work . . . 23
2.5.1 Real-Time Systems and Security . . . 23
2.5.2 Time-Triggered Systems and Security . . . 24
2.5.3 Side-Channel Attacks . . . 24
2.5.4 Mitigation against information leakage . . . 25
2.5.5 Randomization as Attack Mitigation . . . 26
2.5.6 Schedule Randomization . . . 27
3 Detected Vulnerabilities 31
3.1 Classical Security Problems . . . 33
3.2 Assumption of Independent Faults . . . 33
3.3 Attacking Timeliness . . . 33
3.4 Overhead of Security Measures . . . 34
3.5 Vulnerable WCET Derivations and Optimizations . . . 34
3.6 Deterministic Task Execution . . . 35
4 Novel Single Core Attack Vectors 37
4.1 Single Core System Model . . . 37

4.2 Single Core Threat Model 38
5 Single Core Mitigation Strategies 41
5.1 Online Slot-Level Randomization 41
5.1.1 Adaptation of Slot Shifting 42
5.1.2 Example of Capacity Interval Creation 42
5.1.3 Spare Capacity Calculation and Update 43
5.1.4 Online Job Randomization 44
5.1.5 Online Slot-Level Randomization Example 44
5.1.6 Online Slot-Level Randomization Conclusion 45
5.2 Offline Schedule-Diversification 46
5.2.1 Random Schedule Set Selection 46
5.2.2 Schedule Entropy 47
6 Extension to Multicore Systems 49
6.1 Multicore System Model 49
6.2 Multicore Threat Model 50
6.3 Multicore Attack Vectors 52
6.4 Multicore Mitigation Strategies 54
6.4.1 Online Slot-Level Randomization 54
6.4.2 Offline Schedule-Diversification 54
6.4.3 Combined Approach 55
7 Evaluation 57
7.1 ROSACE Case Study 57
7.2 MPARM Runtime Overhead Approximations 58
7.2.1 Approximations for Online Slot-Level Randomization . 59
7.2.2 Approximations for Offline Schedule-Diversification . 60
7.3 Implementation and Runtime Overhead on FreeRTOS 61
7.3.1 Scheduling Algorithm Requirements 62
7.3.2 FreeRTOS Features 64
7.3.3 Scheduler Implementation 69
7.3.4 FreeRTOS Scheduler Implementation Summary 74
7.3.5 Case Study Adaptation 75
7.3.6 Runtime Overhead Measurement Implementation 75
7.3.7 Runtime Overhead Measurement Results 79
7.4 Memory Requirement 84
7.4.1 Offline Schedule-diversification 84
7.4.2 Online Slot-Level Randomization 84
7.5 Solution Space (number of valid schedules) 86
7.6 Schedule Diversity 87
7.7 Determinism Granularity 88

7.8 Discussion 89
8 Conclusion 93
8.1 Final Remarks 95
9 Kurzfassung 97
9.1 Schwachstellen von Echtzeit- und Zeitgesteuerten Systemen . 98
9.2 Entwicklung von Scheduling-Algorithmen zur Vereitelung Gezielter Angriffe 98
9.3 Implementierung und Evaluierung der Algorithmen 99
List of Figures 103
List of Tables 105
Bibliography 107
Lebenslauf 115

1 INTRODUCTION

Computer systems have become increasingly complex and, at the same time, quite small. Nowadays, a system-on-chip the size of a credit card is sufficient for a desktop computer, fully equipped with connectors for peripherals like a keyboard, monitor, and sound. Peripherals may not only be connected via wires, but they also increasingly use radio waves for wireless communication. Computers also communicate with other computers over networks like the Internet, cellular networks, or even satellites. In order to allow communication across a wide range of devices, many different communication protocols exist, e.g. Bluetooth, USB, TCP, HTTP, SSH, CAN, Wi-Fi, and many more. Communication protocols need hardware as well as software support, meaning not only fitting connectors, but also software to decode the signals sent. Modern system-on-chips are required to support this interconnectivity and to serve more than a single purpose. Therefore, manufacturers design and build today's system-on-chips with multiple application areas in mind and integrate different means of networking and connectivity.

Embedded systems profit from the trend that more and more functionality may be fitted on a small device. Besides power and weight, also size is an important and well-known constraint for embedded systems. However, the ever-increasing need for networking ability also affects these systems. The rise of Wi-Fi, Bluetooth, cellular, and IoT protocols was able to transform even embedded devices from isolated units to interconnected devices. Modern cars, for example, consist of dozens, sometimes even more than a hundred interconnected embedded systems (also known as electronic control units) to control everything from the engine and brakes up to infotainment. This trend adds new pathways into these systems which increases their likelihood to become the victim of a cyber attack. Formerly, embedded systems were deemed to be too simple or inaccessible to be a viable attack goal, but nowadays they have to employ more and more sophisticated security measures to protect themselves as well.

Cyber attacks have become a threat for nearly every computer system. Companies, private individuals as well as government agencies register more and more attacks on their systems. The attacker's goals are various in nature: some aim to steal secrets, some intend to disrupt a service while others try to hijack or control the whole system. Their motives vary as well. Not only financial gain, but also political or military advantages might drive these cyber attacks besides other, possibly personal reasons. These attacks have different impact depending on the environment of the computer system they target. While the disruption of an irrigation system might be more of a nuisance to the operator, the disruption of a fire sprinkling system might have fatal consequences for lives and structures it is sup-

posed to protect. When system failure might have catastrophic consequences such as car collisions, airplane crashes, or nuclear meltdowns, such a system is deemed safety-critical.

Passenger transportation is only one of the safety-critical environments that rely on embedded systems. In areas where timing is paramount, embedded systems are often realized as real-time systems. Real-time systems provide timing guarantees, that is, they guarantee that they react neither too early or too late. This property is very desirable for controlling systems like airplanes, railways, cars, power plants, or spacecrafts, which are known to employ real-time systems. Designers of real-time systems have to acknowledge the increased threat to the security of their systems, especially considering their use cases. Attacks on these systems might result in system failures, leading to collisions, environmental hazards, or the loss of human life. Secure and safe system design is thus of utmost importance to real-time systems operating in safety-critical environments.

1.1 Problem Statement

Cyber attacks are on the rise for every kind of computer systems. Real-time systems provide new attack opportunities as they are not only required to work correctly, but also must provide their output neither too late nor too early. If attackers successfully manipulate the timing of a real-time system, they can provoke failure even when the system provides correct output. Time-triggered systems, a subset of real-time systems, exacerbate this risk as they not only provide timing guarantees, but also a fully predictable task schedule. Such deterministic behaviour is a desirable property in terms of security, dependability and system certification. However, this determinism also poses a risk. Adversaries may exploit predictable system behaviour to launch targeted and stealthy attacks. The ability to predict the exact timing of task execution allows adversaries to fine-tune their attacks to their victim task.

Before adversaries are able to predict the timing of tasks, they have to infer the schedule of the targeted system. Side-channels are a prominent means to achieve this goal. Recent security incidents like Spectre [37] and Meltdown [45] have shown that side-channels can be quite prevalent in computer systems. Furthermore, time-triggered schedules consist of only a few bytes of information, so even side-channels with very low bandwidth are sufficient for the adversaries' goal. Thus, even when the time-triggered schedule is not publicly available information, it may be inferred. Information about the schedule of the victim task allows adversaries to only attack when they are sure to hit their victim or at the most favourable opportunity. This strategy increases the likelihood of success and decreases the likelihood of detection by security measures, allowing for more sophisticated and successful attacks.

To mitigate this risk, we present a scheduling strategy employing randomization to prevent predictions about task execution. Randomizing the points of time when tasks execute may however lead to deadline misses of the randomized task or other tasks in the system. Therefore, we developed a mitigation strategy for time-triggered systems that does not violate any of its tasks' timing constraints. Besides the predictable schedule of time-triggered systems, we also detail other vulnerabilities that attackers of real-time systems

might exploit. We especially consider vulnerabilities inherent to the very properties that characterize real-time systems.

The next chapters provide a deep dive into the attacker's means and present ways to harden system security. With this thesis, we aim to contribute towards minimizing the attack surface of real-time and time-triggered systems in order to support the development of more secure and dependable systems.

1.2 Contribution of Thesis

In this thesis, we explore how an adversary is able to maliciously exploit the very properties of real-time and time-triggered systems to mount an attack, how to mitigate such attacks, and how our mitigation strategies perform in terms of runtime overhead, memory cost, solution space and determinism granularity.

1. **Vulnerabilities of real-time and time-triggered systems**
   First, we detail several vulnerabilities specific to real-time and time-triggered systems. While these system inherit all "classical" security problems of (embedded) computer systems, novel vulnerabilities arise from the additional constraints in timeliness and determinism that these systems must fulfill. Every additional constraint constitutes a possible vulnerability that an attacker may exploit as the violation of these constraints means the system deviates from its specified behaviour. We cover vulnerabilities arising from timeliness constraints, worst-case execution derivations or optimizations based on system behaviour, and deterministic execution patterns.

2. **Mitigation strategies for attacks exploiting deterministic task execution**
   Second, we introduce security strategies to mitigate attacks exploiting the deterministic execution pattern of time-triggered systems. For single-core systems, we develop two mitigation strategies that prevent attack vectors relying on deterministic task execution. The first approach, online slot-level randomization, analyses the task set offline to define the execution window for each job. As long as a job executes within the timing constraints presented by its execution window, it does not miss its deadline. With online slot-level randomization, we present a scheduling algorithm that impedes an attacker's predictions by randomizing jobs within their execution windows. By adhering to timing constraints through an online bookkeeping mechanism, our algorithm guarantees that no job misses its deadline. The second approach, offline schedule-diversification, constructs valid schedules offline and stores a diverse set of schedules. At runtime, it randomly switches to another schedule at hyperperiod end. We extend both mitigation strategies to multicore systems and also propose a third mitigation strategy which is a combination of the former two.

3. **Case study and evaluation**
   Finally, we evaluate our mitigation strategies in terms of runtime overhead, memory cost, solution space and determinism granularity. Based on previous work, we approximate runtime overhead for both mitigation strategies with measurements from

an earlier case study on a simulator. Using the ROSACE case study, we implement both mitigation strategies on FreeRTOS, measure the runtime overhead as well as memory cost and compare the runtime measurements against the approximations as well as against time-triggered scheduling without the mitigation strategies. We calculate the available solution space for randomization mathematically, that is, we calculate the number of all possible permutations of the schedule the randomization algorithm is able to explore. After discussing the topic of schedule diversity for the set of offline calculated schedules, we show the consequences of randomization in terms of determinism granularity.

1.3 Outline

The rest of this thesis is structured as follows:

Chapter 2
In this chapter, entitled "Background", we introduce the necessary technical background and related work. We explain concepts of real-time systems, scheduling and real-time operating systems and show the fundamentals of fault-tolerant and dependable systems. Time-triggered systems and security are introduced as well and build a bridge to the following related work section where we delve deeper into security-related topics, such as side-channel attacks, moving target defense, attack mitigation and schedule randomization.

Chapter 3
In this chapter, entitled "Detected Vulnerabilities", we motivate why the search for security vulnerabilities remains important and why real-time systems also suffer from those. They range from classical security problems most (embedded) systems share to those tailored to their real-time properties. We introduce novel security vulnerabilities we found in time-triggered as well as real-time systems.

Chapter 4
In this chapter, entitled "Novel Single Core Attack Vectors", we present a novel attack vector for single core time-triggered real-time systems exploiting one of the security vulnerabilities introduced earlier. First, we detail our system assumptions in the system model, then lay out our assumptions on the attacker's capabilities and how they conduct an attack on a single core real-time system.

Chapter 5
In this chapter, entitled "Single Core Mitigation Strategies", we present two novel strategies for single core time-triggered real-time systems to mitigate the attack vector presented earlier. We extend the concept of schedule randomization to time-triggered real-time systems and present online slot-level randomization and offline schedule-diversification, our two different mitigation strategies, in detail.

Chapter 6
In this chapter, entitled "Extension to Multicore Systems", we extend our threat analysis and mitigation strategies to multicore systems. First, we detail our system assumptions in the system model and then lay out our assumptions on the attacker's capabilities. Based on the threat model, we present three different attack vectors targeting multicore time-triggered real-time systems by exploiting one of the security vulnerabilities introduced earlier. We present three novel strategies for multicore time-triggered real-time systems to mitigate these attack vectors by extending the single core randomization strategies to multicore systems. Lastly, we present a combined approach of both mitigation strategies as our third multicore mitigation strategy.

Chapter 7
In this chapter, entitled "Evaluation", we present the case study we use to evaluate our mitigation methods. First, we approximate runtime overhead based on measurements of related work, then we measure runtime overhead with an implementation of our mitigation strategies on FreeRTOS. We explain the implementation and measurements in detail, discuss the results and compare them to the approximations and pure time-triggered scheduling. The latter serves as a baseline to show how much additional overhead the security measures introduce. Besides runtime overhead, we also discuss the memory requirement of our algorithms and the solution space they are able to explore. We derive a minimal set of schedules to achieve the highest schedule diversity, and show how our mitigation methods influence the determinism granularity time-triggered systems provide.

Chapter 8
In this chapter, entitled "Conclusion", we summarize our findings for vulnerabilities, mitigation strategies and implementation results. We detail the contributions of this work and conclude it with final remarks.

Chapter 9
In this chapter, entitled "Kurzfassung", we give a brief summary of this dissertation in the German language.

PUBLICATIONS & FUNDING

The work presented in this dissertation was supported in part by the *Deutsche Forschungsgemeinschaft* (DFG) through the European projects ByzRT (project no. 4328784) and ReSAC (project no. 490872987). The following is a list of my publications which lay the groundwork for this thesis.

[40] K. Krüger, G. Fohler, et al. “Improving Security for Time-Triggered Real-Time Systems with Task Replication”. In: *24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2018. DOI: 10.1109/RTCSA.2018.00036.

[41] K. Krüger, M. Völp, et al. “Improving Security for Time-Triggered Real-Time Systems against Timing Inference Based Attacks by Schedule Obfuscation”. In: *Work-in-Progress Proceedings of the 29th Euromicro Conference on Real-Time Systems*. ECRTS. 2017.

[42] K. Krüger, M. Völp, et al. “Vulnerability Analysis and Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Systems”. In: *Euromicro Conference on Real-Time Systems*. ECRTS. 2018. DOI: 10.4230/LIPIcs.ECRTS.2018.22.

[43] K. Krüger, N. Vreman, et al. “Randomization as Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Real-Time Systems with Task Replication”. In: *Leibniz Transactions on Embedded Systems* 7.1 (Aug. 2021), 01:1–01:29. DOI: 10.4230/LITES.7.1.1. URL: https://ojs.dagstuhl.de/index.php/lites/article/view/lites-v007-i001-a001.

[70] N. Vreman, R. Pates, et al. “Minimizing Side-Channel Attack Vulnerability via Schedule Randomization”. In: *58th IEEE Conference on Decision and Control (CDC)*. Dec. 2019. DOI: 10.1109/CDC40024.2019.9030144.

2 BACKGROUND

This chapter provides the reader with the necessary technical background needed to follow later chapters. It covers real-time, embedded, and time-triggered systems and explains when these systems become safety-critical. We explain concepts related to dependability, fault tolerance and security that are relevant for this thesis, and discuss related work.

2.1 Real-Time Systems

Several definitions of real-time systems exist. A common understanding of real-time systems is that these systems should react to events in a fast manner. For the definition we use, this is only partially true. We follow Buttazzo's [14] concept who defines real-time systems as systems where the correctness of their behaviour not only depends on the correctness of the computation, but also on the physical time the system produces these outputs. While this can mean that the systems needs to be fast, the definition rather provides an answer to the question: When is the system fast enough? Fast enough becomes a matter of system requirement and, thus, of the environment the system operates in. For example, while F1 drivers need to be able to react in fractions of a second when racing at more than 200 mph to avoid collisions, the average driver has a reaction time of 1 second [23]. This analogy represents the different requirements of the various applications of real-time systems, including flight control systems, nuclear plant control, space missions, and robotics. Non-real-time systems, on the other hand, focus on providing good performance rather than timing guarantees. Table 2.1 summarizes this difference in requirements:

System	Output requirement	Timing requirement
Non-real-time	Correct	Good performance
Real-time	Correct	Satisfy timing constraints

Table 2.1: Different requirements for real-time and non-real-time systems

2.1.1 Tasks

Typically, real-time systems are computer systems running several programs which they call tasks. Operating systems run these programs in terms of processes and threads. A **task** is an abstract concept of processes and threads. In this work, we use the term task as synonym for a process or thread. If tasks repeat periodically, they are called **periodic** tasks. A single instance of a task is called a **job**. Tasks that do not repeat themselves after a fixed period of time are called **aperiodic** tasks. Figure 2.1 shows an exemplary periodic task τ_1 with period T and three instances of the task, jobs $\tau_{1,0}$, $\tau_{1,1}$, and $\tau_{1,2}$.

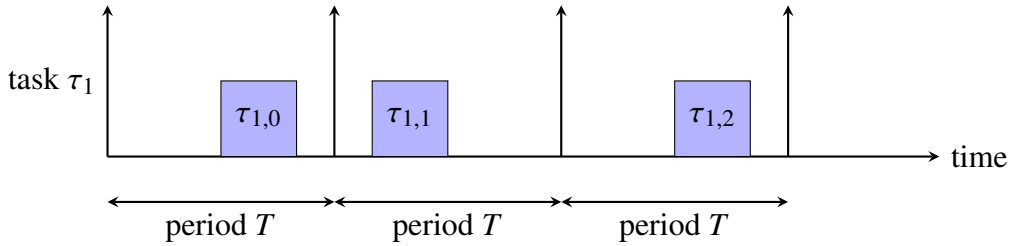


Figure 2.1: Exemplary task τ_1 with period T and jobs $\tau_{1,0}$, $\tau_{1,1}$, $\tau_{1,2}$

In a single core system, the processor can only execute one task at a time. The operating system decides on the next task to execute and dispatches it to run on the CPU. A task in execution is called a **running** task [14]. Other tasks that are able to be executed and that are waiting for the processor are called **ready** tasks. Ready tasks wait in a queue called the ready queue for their turn. The operating system releases new tasks which enter the ready queue when they are able to be executed. In terms of scheduling, whose concepts we explain in Section 2.1.3, the ready queue is often referenced as the ready list. Any task that may potentially execute on the processor, regardless of the processor's current availability, is called **active**. If a task finishes execution, it **terminates**.

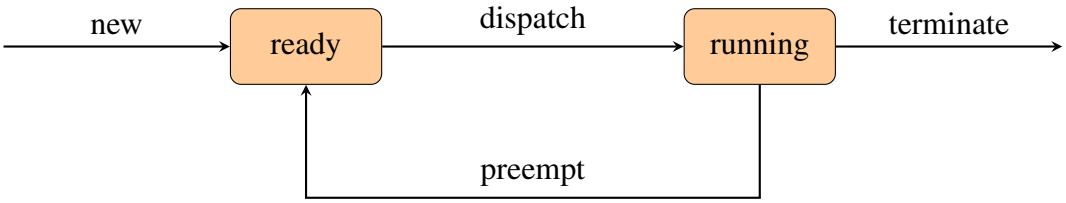


Figure 2.2: Simple process state model

Many operating systems allow the interruption of running tasks. This action is called **preemption** and places the running task back into the ready list to be able to execute another task. Figure 2.2 shows these process states and their respective transitions. Several reasons for preemption exist such as an exception that has occurred and needs to be handled immediately or a newly arriving task which is more important than the currently running task. Real-time operating systems often preempt tasks to be able to guarantee timeliness, especially when the newly arriving task needs to finish soon to fulfill its timing constraint. The next section takes a closer look at timing constraints.

2.1.2 Timing Constraints

Real-time systems must provide their outputs within certain time windows. These windows do not only consist of a deadline, they also have a point in time where they start. Thus, real-time systems must provide their outputs neither too late nor too early, but within these windows. These time windows translate to timing constraints for task. The **release time** and **deadline** of a task mark the start and end of its time window, respectively. Another term for the release time of a task is the **earliest start time**. Both terms also mark the point in time the task is added to the ready list, assuming there are no other constraints like precedence (where it has to wait for another task to finish). If the deadlines of a periodic task coincide with the end of its periods, tasks are said to have **implicit deadlines**. Otherwise we speak of **explicit deadlines**, i.e. they need to be explicitly mentioned and cannot be derived from the task's period. We call deadlines **relative** if they are stated as an offset to another value, e.g. a fixed offset to the task's period. Otherwise we call deadlines **absolute**, i.e. when they refer to a certain point in time. Figure 2.3 shows a task with its release time, explicit deadline and period. In case of an implicit deadline, it is sufficient to only present the release time as Figure 2.4 demonstrates.

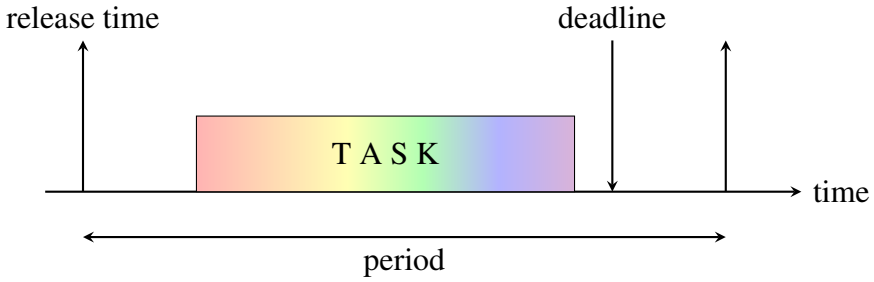


Figure 2.3: Exemplary task with explicit deadline

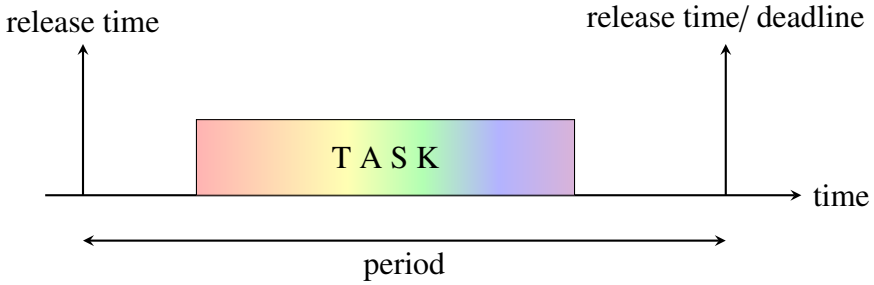


Figure 2.4: Exemplary task with implicit deadline

Each task in a real-time system has several parameters, typically including its period and **worst-case execution time** (WCET). The WCET provides an upper bound on the execution time of a task. Real-time systems need these bounds on execution time to be able to guarantee that, even in the worst case, tasks meet their deadlines. Two different classes of methods exist to derive worst-case execution times: **static analysis** and **measurement-based methods** [73]. Measurement-based methods measure execution time for different

executions of the task, either on the target hardware or on a simulator. However, the state space is typically too large to exhaustively explore all possible execution scenarios of a task. Thus, only a subset of the possible execution times a task can have is measured. While using this method and adding a safety margin on the measured WCET is common and cheap to do, measurement-based methods do not give safe bounds on WCET. Static analysis, on the other hand, does not execute code on hardware or a simulator. Instead, it analyses the code itself together with an abstract model of the system to obtain upper bounds on WCET. These bounds are safe (as long as the model is correct), which is especially important for safety-critical real-time systems which we introduce later in this chapter.

Real-time systems are called **timely** if they fulfill the timing constraints of their tasks. The next section explains how real-time systems fulfill these timing constraints.

2.1.3 Real-Time Scheduling

Scheduling solves the problem of assigning resources to tasks in order to complete these tasks while fulfilling their constraints [14]. In particular, real-time scheduling assigns tasks in a way that they fulfill their timing constraints. The most prominent resources considered for scheduling are processors, i.e. available CPU time, and networks. For this thesis, we consider scheduling with respect to available CPU time and under the consideration of timing constraints. The scheduler, who is in charge of scheduling tasks, is often implemented as part of the operating system. It solves the scheduling problem using scheduling algorithms. Part of the scheduler is the **dispatcher**, whose task is to allocate the CPU to the task that has been selected by the scheduling algorithm. Several classes of scheduling algorithms exist which we detail below. Buttazzo [14] provides the following definitions for different scheduling algorithm classes:

- **preemptive** and **non-preemptive**:
  In preemptive algorithms, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy. In non-preemptive algorithms, a task, once started, is executed by the processor until completion. In this case, all scheduling decisions are taken as the task terminates its execution.

- **static** and **dynamic**:
  Static algorithms are those in which scheduling decision are based on fixed parameters, assigned to tasks before their activation. Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system evolution.

- **offline** and **online**:
  A scheduling algorithm is used offline if it is executed on the entire task set before tasks activation. The schedule generated in this way is stored in a table and later executed by a dispatcher. A scheduling algorithm is used online if scheduling decisions

are taken at runtime every time a new task enters the system or when a running task terminates. In this work, we extend the definition of online scheduling algorithms generally to scheduling decisions taken at runtime, not only on task termination or activation.

Utilization is a measure of how much processor time is spent on the execution of the task set. Consider a task set of n tasks with WCETs C_i and periods T_i. The utilization of this task set is given by:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.1}$$

For single core systems, any value below 1 means there is more processor time available than the combined WCETs of all tasks. Any task set with a utilization above 1 is not schedulable on a single core system as it requests more processor time than is available. A task set is called **schedulable** if there exists at least one scheduling algorithm that is able to produce a feasible schedule [14]. A schedule is called **feasible** if all tasks can be completed according to their constraints.

Scheduling on multicore systems allows for utilization greater than 1 as more cores are available for task execution. However, multicore scheduling is much more complex than scheduling on a single core as cores contend with each other for resources like caches, bandwidth, or memory. The resulting interference increases worst case execution times and complicates the WCET analysis. Multicore scheduling also adds another dimension to the task allocation problem. Not only has multicore scheduling to decide on the time when tasks should execute, but also on which core they execute. Two classes of multicore scheduling exist: partitioned and global scheduling. In **partitioned scheduling**, tasks are statically assigned to cores and that allocation does not change during runtime. This approach simplifies runtime scheduling. On the other hand, **global scheduling** allows tasks to migrate to other cores during runtime. While global scheduling introduces more complexity, it also allows for more flexibility and easier adaptation to system changes like core failures or overload situations.

Overload situations happen when tasks request more processor time than the processor has available [14]. While system designers avoid overloads at design time, tasks might overrun during runtime and cause an overload situation. An **overrun** is a situation in which a task exceeds its worst-case execution time. This might occur when the derived worst-case execution time bound is not safe, e.g. when the bound was derived using measurements. An overrun might also occur when a task is activated before its expected arrival time. Overload and overrun situations might also be the result of failures. An overload may occur when a partial failure of the system leads to a task reallocation across cores. An overrun may occur when a partial failure of the system leads to slower execution, e.g. due to slowed processing of requests. We will look into details of failures and fault tolerance in later sections of this chapter.

2.1.4 Real-Time Operating Systems

The real-time scheduler typically is part of an operating system built specifically for real-time computing, called a real-time operating system (RTOS). RTOSs have a set of unique requirements that distinguishes them from non-real-time operating systems in order to fulfill real-time requirements. These requirements can be separated into five general areas [66]:

1. Determinism
2. Responsiveness
3. User control
4. Reliability
5. Fail-soft operation

We will explain each of these terms in the context of real-time operating systems in the remainder of this section.

Determinism.
An RTOS performs operations at fixed, predetermined times or within certain time intervals and therefore provides a certain degree of determinism. Processes inside the RTOS compete for resources and processor time. External events as well as timing (e.g. new task releases) influence the amount of process requests for resources and processor time. Therefore, the RTOS needs to service these requests with a certain speed and also needs to have sufficient capability to handle these requests within their timing constraints. One important measure of the determinism an RTOS provides is the upper bound on the maximum delay from the arrival of an interrupt until it is serviced.

Responsiveness.
Whereas determinism in the context of real-time operating systems expresses the delay until an interrupt is acknowledged, responsiveness stands for the time the RTOS needs to service the interrupt. Together, determinism and responsiveness express the response time of the RTOS to external events. The response time of a system is critical in the real-time domain as these systems must meet stringent timing constraints.

User control.
Users need to have finer control over the RTOS than in non-real-time operating systems. They need to be able to choose scheduling policies or define their own and assign priorities to processes. More examples of finer control include, but are not limited to: define characteristics of paging and swapping, which processes should remain in main memory, and which rights processes with a certain priority have.

Reliability.
Real-time systems need to operate in a timely manner. They are only able to provide this service if they function correctly. Therefore, they need to operate in a reliable manner. Reliability is defined as the probability of correct service of the system up to a certain point in time.

Fail-soft operation.
This characteristic refers to systems failing in a way that they preserve as much capability and data as possible. Real-time systems must be able to address failures during runtime while remaining operational. Even if they experience failure, they need to be able to fulfill timing constraints, especially in an environment where they are able to impact safety. A missed deadline or degradation of performance can mean harm to people. Fail-soft operation is closely related to the matter of reliability and, ultimately, dependability and fault tolerance. We will explore these matters in further detail in in the next section, Section 2.2.

Figure 2.5: FreeRTOS logo [8]

FreeRTOS is an example of a real-time operating system targeting microcontrollers and small microprocessors [8]. Figure 2.5 shows its logo. FreeRTOS is open-source, features a kernel and a growing number of add-on libraries. Currently, it supports more than 40 different microcontroller architectures, including ARM and RISC-V platforms. The kernel itself is small in size as its typical kernel binary image comprises 6K to 12K bytes. It features support for low-power modes as well as a small memory footprint to be suited for its intended application area: embedded systems.

We use FreeRTOS to implement our mitigation strategies and evaluate them with a case study. Our evaluation includes their runtime overhead and memory cost on FreeRTOS. We present our implementation, evaluation and their results in Chapter 7.

2.2 Fault-Tolerant and Dependable Systems

Systems may not behave as expected during runtime. Several reasons exist for undesired behaviour, ranging from malfunctioning components, for example due to ageing processes, to failures due to malicious influence of adversaries. The rising complexity of systems also complicates the task to construct systems in a way that they behave correctly and continue to do so. A system which shows a high probability of behaving according to its specification is called **dependable** [67]. If we want to assess the degree of dependability a system has attained, we use the following four metrics: reliability, maintainability, availability, and safety [67].

Reliability.
As introduced earlier, reliability is defined as the probability of correct service of the system up to a certain point in time. In other words, reliability is a measure of the continuous delivery of correct service. Commonly, the mean time to failure (MTTF) or the mean time between failures (MTBF) express reliability. Another notion of reliability is the failure rate probability. For safety-critical systems, a typical figure of the failure rate probability is 10^{-9} failures per hour which translates to one failure every 114155 years. Figure 2.6 shows exemplary uptimes and downtimes during the lifetime of a system. Time interval t_1 denotes the time to fail until the first failure.

Maintainability.
Maintainability is the measure of the time to restoration of correct service. Generally, this means the amount of time it needs for repair, which is typically expressed with the mean time to repair (MTTR). Figure 2.6 shows exemplary uptimes and downtimes during the lifetime of a system. Time interval t_2 denotes the time to repair the system from the first failure until the system is up again.

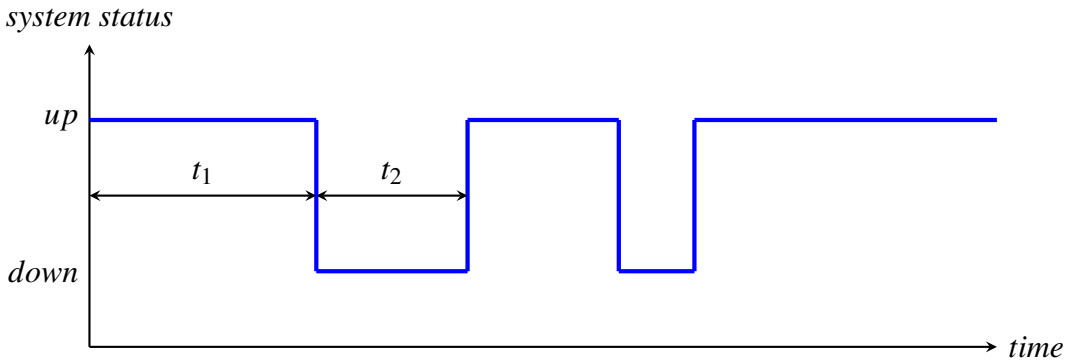


Figure 2.6: Exemplary uptime and downtime during system lifetime

Availability.
Availability is the measure of the delivery of correct service with respect to the alternation between correct and incorrect service. Following this definition, availability can be expressed as:

$$\frac{MTBF}{MTBF + MTTR} \tag{2.2}$$

The result of this equation is a percentage that states the amount of time a system is available, i.e. working correctly. Figure 2.7 shows typical availability requirements for different systems:

Example Component	Availability	Downtime/year
Maintained PC	99%	≅ 4 days
Cluster	99.9%	≅ 1 hour
Embedded Systems	99.999% to 99.99999%	≅ 5 minutes to ≅ 3 seconds

Figure 2.7: Downtime per year for several example components [67]

Safety.
Safety is defined as the degree to which a system, upon failing, does so in a non-catastrophic manner. In other words, safety gives a conditional probability of a failure not being catastrophic when it occurs. Catastrophic consequences mean that the system might not be able to recover from this failure or that the environment, possibly including human lives, is in danger. Systems where failures might have catastrophic consequences are called **safety-critical**. Prominent examples of safety-critical systems are cars, trains and airplanes. Failures that do not have such severe consequences often impact the quality of service provided by the system instead. Safety-critical systems might have critical tasks running alongside non-critical tasks. For example, while the brakes of a car are considered safety-critical, its infotainment system is not.

2.2.1 Fault, Error, Failure

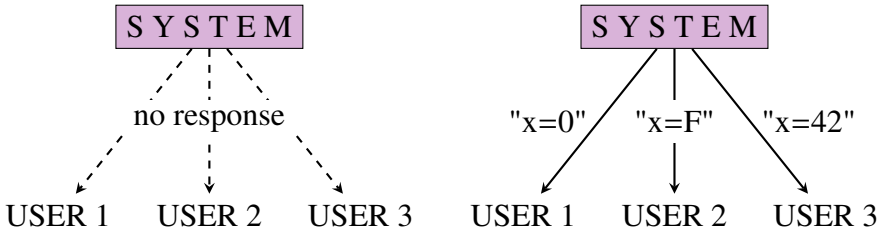
Faults, errors and failures are threats to dependability. The internal or external cause of a failure is called a **fault** [67]. Faults remain dormant until they are activated. A bit flip in memory or a defect in a file system disk record are examples of dormant faults. These faults become active when the corresponding memory region or the defect record is accessed. Their corrupted values then exist as an **error** in the state of the system. A failure occurs as the externally observable effect of the error and may propagate to different subsystems. For example, when the systems reads the corrupted value, a failure of the file system occurs. If this failure propagates, it might appear as strange characters on a screen. Thus, a **failure** occurs when the delivered service of the system deviates from its specification [6]. The

failure of a low-level subsystem may cause a fault at a higher-level subsystems and thus propagate to other subsystems as the screen mentioned earlier. Figure 2.8 shows this chain.

Fault ⟶ Error ⟶ Failure ⟶ Fault ⟶ ···

Figure 2.8: Fault, error, failure with propagation

Faults are classified according to several fault classes [6]. Persistence is one of these fault classes. Faults in the persistence class might either be of **transient** nature, which means they do not persist and are only temporarily present, or of **permanent** nature, which means the fault persists. We also consider intent for fault classification and define **malicious** faults as the result of harmful (human) intent and **accidental** faults as their opposite, meaning that they were not intentionally introduced. Two other types of faults are important for the scope of this thesis: crash and Byzantine faults. **Crash faults** occur when the system completely stops operating [67]. **Byzantine faults** manifest in different users receiving different service (e.g. output). Some users may even receive correct service when the system experiences Byzantine faults.



(a) Crash fault

(b) Byzantine fault

Figure 2.9: Systems tells value of x while experiencing crash and Byzantine fault, respectively

Figure 2.9 shows how a system responds to a request for the value of a variable x while experiencing a crash and Byzantine fault, respectively. The systems provides the response to three users who could also be other systems working with this data. In case of a crash, as shown in Figure 2.9a, there is no system response at all. However, Figure 2.9b shows that a Byzantine fault leads to different answers for each user. When a Byzantine fault occurs, it is hard to tell which user received a correct answer – the system might also not have provided a correct answer at all.

Failures, on the other hand, may be distinguished in two different domains: the content and the timing domain [6]. **Content failures** occur when the content of the information the system or service delivers at an interface deviates from implementing the system function. **Timing failures** occur when the time of arrival or the duration of information the system or service delivers at an interface deviates from implementing the system function. Thus, timing failures may be early or late, analogous to violated timing constraints when tasks execute too early or too late.

2.2.2 Fault Handling and Fault Tolerance

Faults may occur during system runtime for a variety of reasons: components age, moving parts wear and might tear, software bugs might have not been caught during development, electric crosstalk may cause interference, or radiation may introduce bit flips in memory. Dependable systems thus need to be able to handle faults to keep their dependability property. Means that are necessary to achieve this goal do not only include fault prevention and fault removal, but also fault tolerance [6]. **Fault prevention** generally happens during system development, e.g. by using robust programming languages, adhering to standards and guidelines, and maintaining best practices in development processes. The goal is to prevent the occurrence or introduction of faults. **Fault removal** aims to reduce the number and severity of faults and happens during development as well as during the use of a system. During development, validation and verification, including testing, are common methods to remove faults. During use, system maintenance is used to remove faults. In both cases, we first need to identify faults to be able to remove them.

Fault tolerance is the ability of a system to avoid failure in the presence of faults and a very important property for any dependable system. Error detection and fault recovery are both techniques to achieve fault tolerance, which detect the error caused by the fault and then take corrective maintenance. However, this techniques are costly in terms of time. Therefore, in real-time system they are typically combined with **replication** and **fault masking**. Replication concerns the use of replicas which might be exact or diversified copies of hardware or software. Diversification in this context means a copy of a component who produces the same outputs given the same inputs, but whose implementation or design differs. This technique prevents faults that exact copies would share. Full replication of a system is rare, but replication is common for safety-critical components, be they mechanical parts or tasks. The use of several replicas enables the system to mask faults. For example, three replicas of a task should compute the correct distance to an obstacle in the path of a vehicle. When one task experiences a fault, its result will differ from the two replicas working correctly. If the majority of the replicas decides on the result, however, the system is still able to provide a correct output. Figure 2.10 illustrates this example and shows how the majority vote masks the output of the faulty replica.

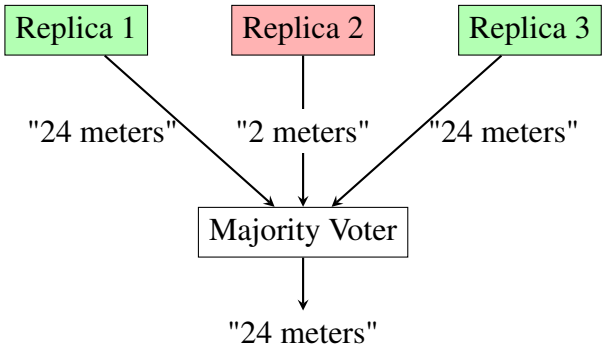


Figure 2.10: Majority vote of three task replicas which masks the faulty output of replica 2

The first step in designing a fault-tolerant system is to assess the faults it experiences. The number and classes of faults a system needs to tolerate is defined in its **fault model**. Thus, the fault model tells the maximum number f of a certain fault it might experience at a certain point in time. We decide on the required number of replicas based on this information. In case of a crash fault, as introduced in Section 2.2.1, the system does not provide any output. Therefore, we only need $f+1$ replicas as only one correct replica needs to provide the output. However, in the case of a Byzantine fault, the number of replicas needed is much higher. Byzantine faults require $3f+1$ replicas to be able to tolerate f faults at the same time.

2.3 Time-Triggered Systems

Depending on the event that triggers actions within the system, real-time system are divided into event-triggered and time-triggered systems [39]. In **time-triggered systems**, the progression of the clock initiates all actions, whereas in **event-triggered systems**, any significant event other than the clock tick initiates actions. Example events for event-triggered systems are the reception of a message or the termination of a task. These events trigger an interrupt to signal that the event has happened and the system then decides on adequate actions. Inherently, event-triggered systems rely on dynamic scheduling to be able to respond to events when they occur.

Contrary, time-triggered systems typically employ static scheduling. The clock tick initiates any action in the system, communication as well as processing activities. These actions are periodic in nature. All parts of a time-triggered system, especially in the case of a distributed system, must share the same time base. Therefore, they synchronize to a universal abstract notion of time known as **global time**. Global time is and needs to be available to every part of the systems. The finest granularity of a time-triggered system is the distance between two consecutive ticks of its global time, called a **slot**.

Scheduling in a time-triggered system builds upon the timing abstraction slots provide. We express scheduling parameters like the worst-case execution time or period in terms of slots. In every slot, only one task executes. The time-triggered system enforces strict separation in the timing domain between tasks, also known as temporal isolation. It invokes the scheduler at the beginning of each slot. In a purely time-triggered system, the schedule is static and defined offline. Thus, the scheduling overhead is low, as the scheduler does not make complex decisions at runtime. Tasks, however, rarely execute for their worst-case execution time. In a static time-triggered schedule, slots reserved for a task are not given to other tasks when the task finishes earlier. Several methods exist to add flexibility to time-triggered schedules, e.g. slot shifting allows the incorporation of aperiodic tasks [27]. However, for this thesis, we focus on time-triggered systems with a fixed task set and a static schedule.

2.4 Security

Security and dependability are interrelated issues that we need to address to achieve a safe system. A dependable systems needs to be secure, and, vice versa, only a secure system can become dependable. While dependability focuses on faults and failures, security deals with attackers, also called adversaries. Attackers cause faults and failures in the system, when they are successful, however, the difference lies in their (malicious) intent. Past and recent attacks on systems have shown their threat to safety, including computer viruses, cars leaking owners' data to other users [1], and even a jeep being forcibly stopped on the highway [31]. **Security** is defined as the absence of unauthorized access to, or handling of, system state [6]. Three attributes describe important properties a secure system must exhibit: availability, confidentiality, and integrity.

Availability.
Availability has already been introduced with respect to dependability and shows the interrelation between security and dependability. From a security perspective, availability stands for the prevention of unauthorized withholding of information. Not only the system and its services, but also its information needs to be available.

Confidentiality.
Confidentiality is defined as the prevention of unauthorized disclosure of information. Information needs to be available and the system needs to control the access to the information. Unauthorized third parties, e.g. attackers, must not get access to information, especially information critical to system safety.

Integrity.
Lastly, integrity describes the prevention of unauthorized amendment or deletion of information. While systems must not only control access to information, they must also control who is allowed to modify information and prevent unauthorized modification.

These three properties demonstrate that the prevention of attackers from reaching various goals is essential to security. Therefore, security measures and strategies are defense strategies at heart and aim to counter attacks. Attackers often try to find and exploit vulnerabilities of a system. A **vulnerability** is defined as a deficiency present in the design or operation of a computer system that can lead to a security incident [39]. Spectre [37] and Meltdown [45] are two examples of prominent and widespread security vulnerabilities. The successful exploitation of a vulnerability is called an **intrusion**. Each pathway into a system and the means used by the attacker constitutes an **attack vector**. The target of the attacker is usually called the **victim** which does not mean the victim is a person. Victims may also be subsystems, tasks or certain functionality.

For security analysis, the assumptions and properties of a system under attack are described in its **system model** whereas the assumptions and means of the attacker (including attack vectors) are listed in the **threat model**. A multitude of different attack vectors exists.

Two attack vectors are important for this work: denial-of-service (DoS) attacks and side-channel attacks. **DoS attacks** aim to make the victim to violate the availability requirement of systems, i.e. they aim to make the system unavailable. For example, a web server might be flooded with requests until it can no longer serve all these requests. Another example for a DoS attack is a jammer in a wireless network setting that blocks the network's frequencies by emitting a signal with higher power. **Side-channel** attacks are attacks that exploit the system implementation and/or its identifying properties [58]. A side-channel is any unintended communication channel within the system. As such, it is characterized with a corresponding bandwidth. Spectre [37] and Meltdown [45] have shown that such side-channels may persist and remain undetected for many years. Side-channels can exist in hardware as well as software [53].

In the related work section, we delve deeper into these topics, explain different side-channels, mitigation strategies, and present work already done in these fields.

2.5 Related Work

In this section, we position our thesis against existing work, discussing time-triggered systems and security, side-channel attacks, moving target defense, and schedule randomization as attack mitigation.

2.5.1 Real-Time Systems and Security

Over the last two decades, security became an increasingly important research topic in the area of the real-time systems. M. Hasan et al. [32] analysed the majority of research on real-time system security during the last decades to identify and classify the various categories these works fall into. Figure 2.11 shows their taxonomy as hierarchical graph.

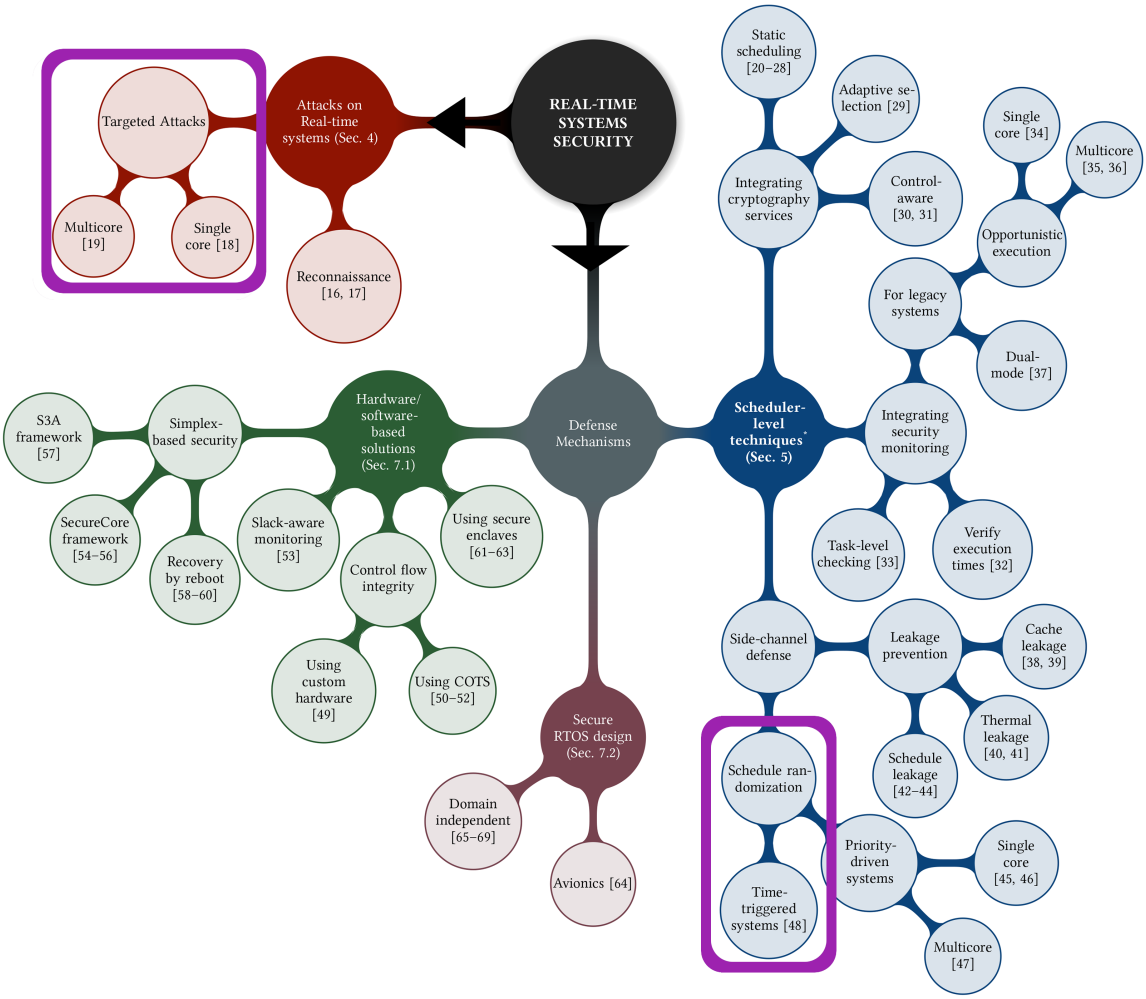


Figure 2.11: Taxonomy of research on real-time systems and security according to [32]. Purple frames indicate the topics this thesis addresses. References in the figure refer to [32].

Two main categories for real-time system security exist: research on attacks and defense mechanisms. Attacks on real-time system either consider targeted or reconnaissance

attacks to gain knowledge of the system in question. Reconnaissance attacks intend to violate the confidentiality of secure systems. As Figure 2.11 shows, there exist a multitude of defense mechanisms against possible attacks in the real-time domain. Besides secure real-time operating systems design methods, also hardware/ software-based solutions are possible. The majority of works deals with scheduling-based techniques as pointed out in [32]. Each mechanism has certain assumptions on the attacker. In this work, we assume the attacker gained knowledge of the schedule through reconnaissance, i.e. the attacker inferred this information over side-channels. We identify points in time most suited for a targeted attack on single and multi-core systems. In order to avoid these targeted attacks, we devise scheduler-based defense mechanisms which employ schedule randomization for time-triggered real-time systems.

2.5.2 Time-Triggered Systems and Security

Several bodies of work exist that tackle the topic of security in time-triggered real-time systems. Wasicek [72] analyses security aspects of time-triggered systems and enhances the time-triggered architecture with security services. He discusses dependability mechanisms of the time-triggered architecture under the aspect of security, considering the following attack vectors: flooding, address spoofing, software attacks, physical modifications, Byzantine attacks, and temporal attacks. These attacks focus on the network level. His proposed security architecture adds additional measures to deal with shortcomings of the dependability mechanisms under security aspects. All of these properties and measures are important to strengthen system security (albeit possibly introducing an overhead, as Wasicek acknowledges) and we endorse taking the necessary steps to strengthen system security. In our work, we assume that these measures might be faulty, that is, a vulnerability is present in the system through which an attacker is able to gain unprivileged access.

Skopik et al. introduce a security architecture for time-triggered communication in [63] which adds device authentication, secure clock synchronization and application level security. Wasicek et al. [71] investigate the security of time-triggered transmission channels and show how an authentication protocol secures these channels without violating timeliness properties. In our work, we do not consider intended communication channels for inferring timing information, but instead focus on covert or side channels and the implication of attackers learning timing information to coordinate their attacks.

2.5.3 Side-Channel Attacks

Time-triggered systems execute offline constructed schedules which have already accounted for task parameters like deadline and period. For other scheduling algorithms like Rate Monotonic Scheduling, these task parameters are sufficient to be able to reconstruct the schedule if the system only consists of periodic tasks. This critical timing information may be leaked over side-channels. Side-channel attacks are based on attackers gathering knowledge about a system, and in turn exploit this knowledge to influence system behaviour [2, 49, 57, 64, 65].

For example, Liu and Yi [47] show how an attacker is able to infer task parameters under fixed-priority and rate monotonic scheduling. They infer the number of tasks, the period, and execution time of each task by tracing busy and idle times of the system. Another team of researchers infers the speed of a car engine from the frequency of execution of its control task by analysing the scheduling sequence [46]. Chen et al. [16] acknowledge this threat, especially in the context of real-time systems, and show how an unprivileged task learns the precise timing behaviour of its critical victim task by observing its own execution intervals.

The above mentioned works fall into the category of scheduling side-channel. The cache may prove to be another side-channel. Mushtaq et al. [53] provide a survey over a decade of cache-based side-channel attacks. They review various types of cache-based software side-channel attacks and show the vulnerability of cryptosystems, in particular RSA, to these attacks, but also discuss detection and mitigation strategies.

In this work, we build upon this strong evidence that timing information may be leaked over side-channels and assume that the attacker is able to infer timing information. Knowledge of timing information allows for pinpointed attacks against the victim. We devise mitigation strategies to prevent an attack based on the knowledge of the scheduling decision.

2.5.4 Mitigation against information leakage

Several security solutions exist which prevent information leakage in real-time systems. For example, Völp et al. [69] prevent timing leaks in fixed-priority schedulers by exploiting the idle task to mask early stops or blocks of a high priority task such that a low priority task always has the same view of the high priority task. Time-triggered systems do not require this modification since two tasks do not coexist in the same time window on the same processor as they have statically assigned time slots based on worst case assumptions. In [52], Mohan et al. focus on the problem of information leakage over shared resources. They define security levels for tasks and prevent undesirable information flow between tasks of different security levels by flushing the resource. Further, they discuss the integration of security constraints into the design of fixed-priority schedulers. In contrast to [69] and [52], we consider time-triggered systems which have no concept of task priority. Additionally, we do not focus on preventing side channels or information leakage. In fact, we assume timing information, in particular task set parameters, may be inferred.

Chen et alii present a framework for single core [17] and multi-core platforms [18] that protects against a posterior scheduling side-channel attack by preventing untrusted tasks from executing right after the victim task. The attacker they consider executes right after the victim and intends to steal, overwrite or corrupt the victim's secret. In our work, we assume the attacker's goal is to launch a precisely timed attack instead. However, our mitigation strategies also prevent this type of attack to work reliably as the attacker is not able to know the schedule in advance. The next sections show how systems can employ randomization as attack mitigation strategy, upon which we build our mitigation strategies.

2.5.5 Randomization as Attack Mitigation

The goal of randomization as attack mitigation is to move the target or victim of the attacker in a manner that is hard or not possible to predict. Different approaches for different situations and parts of the system exist.

Moving target defense covers techniques to enhance the resilience of a computer system by making it more dynamic, random, and/or diverse [55]. Here, 'dynamic' refers to changing the properties or internals of a system over time, 'random' to making an aspect of a system less predictable, and 'diverse' to having different copies of a system with different internal properties. This broad definition includes replicated circuits as used in, e.g. IBM's Launch Vehicle Digital Computer in the Saturn V rocket of NASA's Apollo spaceflight program in the 1960s and early 1970s [21], which enhance fault tolerance. However, we focus on moving target defense techniques to mitigate malicious attacks, in particular with its property of randomness. Figure 2.12 shows different domains to which moving target defense can be applied.

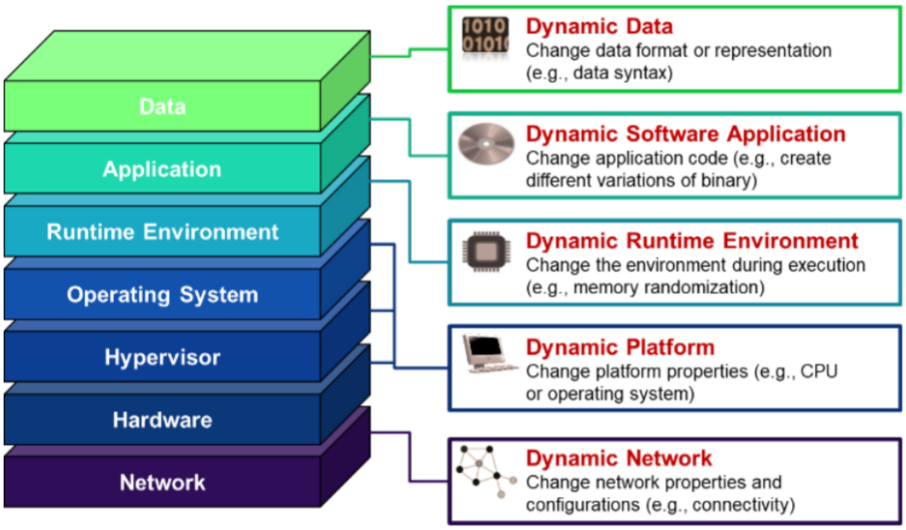


Figure 2.12: Domains of moving target defense compared against a typical compute stack [55].

We present examples of works using randomization as mitigation strategy in different parts of the compute stack [55]. In the dynamic data domain, Cadar et al. [15] present a data randomization technique that uses a random key to mask data values in an application to prevent modification. Franz et al. [30] insert a random number of NOP (no operation) commands into the application during compilation. Their technique influences the dynamic software application domain to strengthen the system against control hijacking exploits. Address-space layout randomization (ASLR) is a prominent technique in the dynamic runtime environment domain. ASLR loads programs or system modules into memory at randomly selected addresses [62]. This approach impedes an attacker trying to predict target addresses to, e.g., exploit a function residing in memory for which the corresponding address must be known, and is particularly useful against code injection or code reuse attacks. Bangalore and Sood [7] present an approach where they periodically exchange server virtual machines with fresh copies to remove an attacker from the system.

Their method falls into the dynamic platform domain which targets underlying platform properties like the operating system, hypervisor or hardware. The remaining domain, dynamic network, is out of the scope of this thesis.

All these methods have in common that they rely on randomization and thus a sufficient amount of entropy. Two assumptions must hold in order for methods based on randomization to be successful:

1. The attacker must not be able to enumerate all[1] possibilities (and thus bypass the moving target defense).

2. The attacker must not be able to predict the randomized values.

True random number generators are a reliable choice for generating unpredictable values. However, also pseudo-random number generators may be used, but they must keep their initial state (e.g. a key or seed) secret, otherwise they are easily compromised. For this work, we assume the randomized values stem from a trusted component and are sufficiently unpredictable for our purpose. We use the advantages of randomization to impede predictions about the schedule. In terms of target moving defense, we reside in the operating system, as the scheduler is part of it, and influence the dynamic runtime environment domain. Our mitigation strategies randomize the point in time when an application executes and thus randomize the schedule. In the next section, we take a look at existing work on schedule randomization.

2.5.6 Schedule Randomization

Real-time systems should exhibit deterministic and repeatable behaviour. Periodic tasks with bounded execution time, i.e. an upper bound in form of their worst case execution time, are easy to integrate into systems with these requirements. Time-triggered systems with an offline constructed schedule take the granularity of determinism [28] even further. For each time slot of a time-triggered system, the task executing is known beforehand. Attackers may use this very quality of real-time and especially time-triggered systems against them. Repeated patterns allow for exploitation by an attacker who can easily determine the time when its victim executes. With this knowledge, the attacker is able to deduce the point in time best suited for an attack.

Schedule randomization methods aim to mitigate this attack vector by interrupting repeated schedule patterns while adhering to the (real-time) constraints of the system. When the attack mitigation method randomizes schedule patterns of a real-time system, care must be taken that randomization does not lead to deadline misses. This often involves some kind of bookkeeping mechanism which tracks leeway in the schedule. We explain this mechanism for our mitigation methods in detail in Chapter 5 where we present a bookkeeping mechanism similar to slot shifting. Yoon et al. [74] use a similar method for

1 An attacker might successfully execute an attack with the knowledge of only a subset of all possibilities. This must also be prevented and is typically done by choosing an order of magnitude for the solution space such that also a subset is not enumerable for an attacker.

fixed-priority real-time systems assigning priorities according to Rate Monotonic Scheduling. They compute the worst-case inversion budget for each task, that is, a bounded amount of priority inversions a task may experience before deadline misses occur. As long as the scheduler does not exceed this budget, all tasks are able to meet their deadlines. They introduce schedule entropy, which is an approximation of Shannon entropy, as a measure for the randomness of different schedules. The more diverse (or different) two or more schedules are, the higher the schedule entropy is. They conclude their work with the claim that they provide security in a measurable way.

Jiang et al. [35] present a schedule randomization method for real-time systems using Earliest Deadline First (EDF) scheduling. They focus on differential power analysis attacks on the Advanced Encryption Standard (AES), a widely used cryptographic algorithm for confidentiality protection. An attacker which uses this attack is able to accurately measure power consumption and analyse it to compute the cryptographic secret key AES uses. This type of attack is successful if the power consumption samples are mostly free of noise and well aligned, which is the case with EDF schedulers as the schedule repeats each hyperperiod. In other words, the randomization of the schedule prevents or, at least, impedes this type of attack. Thus, they devise the SPARTA scheduler which randomly allocates leakage points of tasks, that is, points in time which can potentially leak information about the secret key. The SPARTA scheduler guarantees the deadline constraints of the tasks by only randomizing within a certain time interval based on EDF scheduling and thus uses the same schedulability test as EDF.

In contrast to [35] and [74], we focus on time-triggered systems, which do not have a notion of priority. Also, we advise against using schedule entropy as security metric, see Chapter 5.2.2.

Samaddar et al. [60] present a similar threat model for wireless networks. They consider real-time flows with fixed periods and stringent deadlines over globally synchronized network nodes executing a static TDMA-based schedule. In order to disrupt the repetitive pattern of the schedule, they generate a set of different network schedules offline which they distribute to each network node. All nodes use the a pseudo random-number generator with the same value and seed to randomly select the same next schedule after each hyperperiod. Our offline mitigation strategy also uses a pre-generated set of schedules within which the scheduler randomly chooses the next schedule after each hyperperiod. However, we consider CPU, not network scheduling, and also present an online mitigation strategy which does not rely on offline constructed schedules.

Nasri et al. [54] discuss schedule randomization as defense against schedule-based attacks. They argue that schedule randomization as presented in this thesis is oblivious to potential attacks and hence not able to secure a system. They further claim that randomization may generate schedules with more potentially beneficial situations for the attacker task (e.g. scheduled right before the victim task) than the static schedule. We build our mitigation strategy on the assumption that the attacker is not able to predict randomized scheduling decisions, hence the attacker is no longer able to predict when attacks are effective. While situations may occur in which the attacker would be in a favourable position for an attack, the attacker is not aware of it. This prevents the stealthy attacks we aim

to defend against, as we detail in later chapters. Nasri et al. also claim that their implementation of our online mitigation strategy had a runtime overhead that was too high to schedule tasks. In our evaluation section, we show the low overhead of this mitigation strategy based on an implementation of FreeRTOS and how it is comparable with pure time-triggered scheduling.

3 DETECTED VULNERABILITIES

Most, if not all computer systems have exhibited vulnerabilities at some point, either during development or during operation. Rigorous design and iterative processes in development aim to eliminate vulnerabilities, but over time and with the advancement of technology, possibilities and interconnectivity, the attack surface increases. This chapter motivates the search for vulnerabilities and provides a list of vulnerabilities we found present in real-time systems.

For modern CPUs, the most prominent security vulnerabilities are Spectre [37] and Meltdown [45] which, at the time this thesis was written, still produce new variants and related vulnerabilities. Spectre and Meltdown were made public in 2018 and affected nearly all modern processors at this time. This meant not only processors in personal computers for home and office use were vulnerable, but also commercially used hardware, for example, in servers. Their impact is so severe that security researchers initially did not believe the vulnerabilities to be true [36].

Figure 3.1: Meltdown and Spectre logos [25]

Spectre is, more precisely, a class of vulnerabilities which exploits branch prediction, a common feature on modern processors. Mispredicted branches often leave traces of data, e.g., in the cache which an attacker can use to obtain sensitive data in otherwise inaccessible memory regions. While Spectre exploits the concept of speculative execution in general and branch prediction in particular, Meltdown is a hardware vulnerability. Meltdown allows a process to read memory regions it should not be able to read. In contrast to Spectre, it exploits a race condition present in the instruction processing of many modern processors to circumvent privilege checks. Mitigation of both vulnerabilities requires not only software patches, but also firmware fixes and hardware redesign of many processors of many different vendors. Many mitigation approaches heavily impact performance

which not only meant reluctance when applying patches, but also e.g. Intel offering protection against Spectre as a feature instead of a bug fix, which led to controversy [20]. These two examples show how far vulnerabilities may reach, that they can be present in software as well as hardware and that their mitigation may have a major impact on system performance.

Safety-critical systems are systems which are, in case of failure, directly able to impact the health of their users or cause catastrophic consequences for their environment. Cars are a prominent example of safety-critical systems where more vulnerabilities are discovered with each passing month. In particular, the advances of computers inside cars (ECUs), electronic vehicles, and autonomous driving as well as their growing complexity contribute to a greater attack surface. In 2015, a team of security researchers demonstrated a live hack of a passenger vehicle [31]. They did not physically alter the car and were instead able to carry out the attack remotely over the car's onboard wireless capabilities. The vulnerability they had discovered allowed for a remote exploit through which they were able to send messages over the vehicle's CAN bus [50]. Not only were they able to control AC, dashboard and radio, they were also able to blind the driver with wiper fluid and to stop the engine in the middle of the highway. Moreover, they could wirelessly control the steering, transmission and brakes with this exploit. With this level of access to critical components of the car, malevolent attackers are able to kill people travelling in the vehicle.

Aside from potentially lethal consequences when the security of safety-critical systems is compromised, cars may also disclose personal information. Modern cars log statistical information related to driving performance and vehicle handling and offer to connect to their owner's smartphone, often with a car-specific app. In 2019, the app provided by one major car company for their customers leaked personal information of the car owner to other app users [1].

Besides privacy concerns, car manufacturers also need to ensure that software updates for, e.g, security issues do not affect certified safety-critical parts of the car. A recent McKinsey study from 2020 acknowledges these issues and concludes that "cybersecurity is becoming a new dimension of quality for automobiles" which "will become nonnegotiable in the long run" [13]. The study also lists further software vulnerabilities observed across the entire digital car ecosystem, e.g. a vulnerability in infotainment systems which allowed to gain control of microphones, speakers, and navigation systems, and vehicle data exposed during registration which allowed for remote denial-of-service attacks on the car, to name a few.

The above examples serve the purpose of showing that software vulnerabilities impact day-to-day life and that the security compromise of a system might directly affect safety and the health of its users. While vulnerabilities seem to be inevitable, the first step in protecting against a vulnerability and preventing a security compromise is becoming aware of it. Hence, we dedicate this chapter to detected security vulnerabilities specific to real-time systems. It is an effort to present vulnerabilities inherent to real-time system in general and with a glance at state-of-the-art research. Unfortunately, the list of presented vulnerabilities will not (and probably never will) be complete as we expect more vulnerabilities to emerge in the future, as the past has taught us. The goal of this chapter and thesis in

general is to show how the real-time system's needs as well as essential system properties may be used against the system itself. We endorse future researchers to expand this list and to investigate real-time systems tailored to their needs for vulnerabilities.

3.1 Classical Security Problems

Before we explore real-time system specific vulnerabilities, we also need to consider well-known and existing security vulnerabilities (and their mitigation strategies) outside the real-time domain. Common threats to security are, for example, backdoors, malware, denial of service attacks, unauthorized access, viruses, and side-channels. Real-time systems inherit all classical security problems, especially those for embedded systems if they are implemented as such. While research on security specifically for real-time systems has finally caught on in the last few years, the research body on security for embedded systems and computer systems in general is vast. Depending on the application scenario, many vulnerabilities can be detected and mitigated by keeping up to date with non-real-time-specific security. Of course, we need to consider our system assumptions as well as its requirements to be able to decide on suitable measures.

3.2 Assumption of Independent Faults

So far, the fault models of embedded as well as real-time systems typically assume independent faults. However, adversaries may maliciously trigger faults which are no longer independent from each other. The intent behind an attack, i.e. maliciously provoked faults, contradicts the independence assumption. This violates one of the basic assumptions of the system's fault model. Hence real-time systems that aim for fault tolerance even when under attack, i.e. the ability to tolerate maliciously provoked faults, need to consider dependent faults for their fault model.

3.3 Attacking Timeliness

The purpose of real-time systems is to provide guarantees for timeliness. Hence, any attack on the timeliness of a real-time system constitutes an attack on its fundamental property. This stands in contrast to general-purpose systems where an attack on timeliness may affect the quality of service provided, but not defeat its very purpose. Disrupting the timely behaviour of real-time systems may also have severe consequences for its environment. Real-time systems provide guarantees for timely execution and/or transmission which makes them suitable for safety-critical applications. If applied in a safety-critical environment, timeliness becomes a safety property. Thus, a violation of the timeliness constraints of a real-time system not only defeats its purpose but can also pose as direct threat to safe operation. In the context of safety-critical application areas like avionics, this can mean harm to human life. In conclusion, real-time systems are inherently vulnerable to attacks which aim to disrupt timeliness and, e.g., cause tasks to miss their deadlines.

3.4 Overhead of Security Measures

Time, especially processing time, is a limited resource. Operating systems use the scheduler to give fractions of the total available CPU time to tasks. In case of real-time systems, they ensure that the operating system does not allocate more CPU time to tasks than it has available and prevent overload situations. Otherwise, tasks may miss their deadline and the timeliness violation in turn might threaten system safety as we explain above. For this purpose, the execution time overhead of every task in the system needs to be taken into account and carefully evaluated against timeliness and (quality of) service requirements.

Security measures implemented in software may need their share of scarce CPU time as well. Every task added for security purposes reduces the available CPU time for other tasks running on the same core. Therefore, the real-time system designer has to carefully weigh the CPU resource demand of security measures against the need for these measures in general. While this applies to some degree to computer systems in general, general-purpose systems prioritize average performance over timeliness as they are not designed for the latter. Violating timeliness constraints in general-purpose systems due to, e.g., an overload situation means a degradation in the provided quality of service. However, the same violation might directly lead to unsafe operation in the case of real-time systems. Therefore, the execution time overhead of security measures makes real-time systems inherently vulnerable as they must not exceed their budget of available CPU time. This budget also dictates the upper bound of execution time the real-time system is able to spend on security measures and thus limits them.

3.5 Vulnerable WCET Derivations and Optimizations

Two different classes of methods exist to derive worst-case execution times (WCET): static analysis and measurement-based methods [73]. Measurement-based methods measure execution time for different executions of the task, either on the target hardware or on a simulator. However, the state space is typically too large to exhaustively explore all possible execution scenarios of a task. Thus, only a subset of the possible execution times a task can have is measured. While using this method and adding a safety margin on the measured WCET is common (and cheap to do), measurement-based methods do not give safe bounds on WCET.

Static analysis, on the other hand, does not execute code on hardware or a simulator. Instead, it analyses the code itself together with an abstract model of the system to obtain upper bounds on WCET. These bounds are safe (as long as the model is correct), which is especially important for safety-critical real-time systems. Since abstraction means that information is lost, static methods overestimate the WCET.

Both methods share the same flaw: If they only simulate, analyse or measure intended behaviour of the system, then the derived WCET does not account for malicious behaviour. A system under attack may deviate from its intended behaviour. For example, adversaries may infiltrate a (non-critical) task at runtime. The hijacked task then starts accessing the cache in a way to create maximum interference for the next task's execution, as shown

in [9]. The next task's WCET may not account for this cache pattern if this (malicious) behaviour was not encountered during uncompromised execution and the task may miss its deadline.

Optimizations may increase the severity of this vulnerability. We mentioned earlier that static methods overestimate the WCET. A research body exists to reduce the pessimism of WCETs derived this way. For example, research on abstract interpretation WCET derivation claims that the assumption of cold caches is too pessimistic for a real system and shows methods to achieve tighter and less pessimistic WCET bounds [34], [26]. These optimizations are based on the assumption that tasks behave as intended. As mentioned above, adversaries are able to violate this fundamental assumption. Thus, we have to derive WCETs in a such way that they also account for malicious behaviour and we have to check the impact of performance optimizations on system security.

3.6 Deterministic Task Execution

Deterministic task execution is a desirable, sometimes even necessary system property. At each point in time, the task executing is known. To achieve this, designers of time-triggered systems typically allocate slots to tasks and their jobs before runtime. They solve complex scheduling constraints offline and construct the schedule in advance. This reduces runtime complexity and allows the schedule to be stored as a table such that the runtime scheduling complexity equals a table lookup. The scheduler only needs to start the next task in this table as online calculations and complex decision-making are obsolete using an offline constructed schedule. Moreover, this determinism also allows for easy detection of abnormal behaviour in the time domain. Table 3.1 shows an example of an offline schedule, where numbers below the task ID represent slots.

ID	0	1	1	0	2	...
Start	1	8	20	33	35	...
End	2	11	23	34	40	...
WCET	1	3	3	1	5	...

Table 3.1: Example of an offline constructed schedule

Time-triggered systems realize deterministic task execution by invoking the scheduler after certain time periods, called slots [61]. They separate tasks from each other temporally by employing the above mentioned, offline constructed schedules. Typically, the schedule length equals the hyperperiod and the schedule repeats itself periodically. However, one of the main vulnerabilities of deterministic real-time systems in general and of time-triggered systems in particular lies in this deterministic behaviour. Assume attackers who want to fine-tune their attacks to hit at the right time, for example when a critical task executes. Due to the deterministic nature of the system, it keeps executing the same schedule. If the attackers employ a side-channel, they are able to reconstruct the schedule in reasonable time. Even when the channel has low bandwidth, for example, 1 byte per second, the

schedule can be found out in a matter of a few minutes as schedules for time-triggered systems comprise only a few bytes of information. As we show in Chapter 7.4.1, an offline constructed schedule of a real-world system can consist of just 50 bytes. Through the aforementioned channel with low bandwidth, the attackers would know the schedule within less than a minute and are then able to use the schedule to find the best opportunity to attack their victim.

Therefore, we reason that timing information can be inferred in these systems, and focus on mitigating attacks under this assumption. We dedicate the following chapters on investigating this threat, propose adequate mitigation strategies and evaluate them with a real-world implementation.

4 NOVEL SINGLE CORE ATTACK VECTORS

As systems develop and increase in complexity and interconnectivity, new attack vectors emerge. Security measures are racing with more and more sophisticated attacks, as attackers continue to refine their skills and find new attack vectors. Attacks are typically tailored to a vulnerability of a specific system and its underlying properties. This chapter focuses on single core time-triggered real-time systems and presents novel attack vectors which target the inherent and desired property of these systems: determinism. We show how attackers are able to exploit this property for a more sophisticated type of attack. For this purpose, we present our assumptions on the system with the system model. In the threat model, we detail our assumptions on attackers and how they mount their attack. In a later chapter, Chapter 6, we extend this attack vector to multicore systems with task replication, which allow for an even greater attack surface.

4.1 Single Core System Model

We assume a time-triggered real-time system implemented on a single core platform. A real-time operating system (RTOS) is present, that is, we do not consider bare metal real-time implementations. The real-time system runs safety-critical as well as non-safety-critical tasks. A feasible schedule is constructed offline and then adjusted online according to task set parameters as discussed in Chapter 5. We assume the schedule is correct and that the task set is schedulable. Precautions such as authenticated boot are in place to ensure that the schedule is correctly deployed in the real-time system.

Time-triggered systems and their schedulers rely on a global time base. We consider the global time base protected from attacks, as it is a critical system component. Global time is divided into slots of the same size, which are the granule the scheduler uses for job preemption. Our focus lies on CPU-level scheduling, hence we do not consider time-triggered networks or communication channels. Moreover, we assume a time-triggered system without asynchronous event activation.

Each task τ_i is defined as the tuple $\tau_i = \{C_i, T_i, D_i\}$, where C_i is the worst-case execution time of the task, T_i is the task's period, and D_i is the task's relative deadline. We denote the absolute deadlines of task instances, i.e. their jobs, with d_i. To guarantee the correct execution of the task set, time-triggered systems ensure that time equal to or greater than the worst-case execution time is assigned to each job of each task. This means that the time budget is allocated to the task even when the job completes its execution early. Throughout the rest of the work, we assume that $D_i = T_i$, adopting the implicit deadline task model.

We define the time interval between task release and task deadline as this task's execution window.

4.2 Single Core Threat Model

We assume attackers are able to successfully infiltrate the system through undetected vulnerabilities and compromise non-safety-critical tasks. In particular, we assume that safety-critical tasks are sufficiently shielded against direct attacks which requires attackers to find a pathway through non-safety-critical tasks. Less stringent evaluation requirements make non-safety-critical tasks, including non-real-time tasks, primary targets for attackers. Firewalls and gateways protecting safety-critical tasks in autonomous vehicles and planes support this assumption. Even though we assume intrusion detection [19, 48, 51, 76], hardening mechanisms and other defenses against common attack vectors (e.g. DoS attacks) are in place, we acknowledge that these techniques are imperfect. Past research and experience have shown that attackers might find complex pathways into systems nonetheless and compromises may go undetected. In our case, we restrict the compromise to non-safety-critical parts of the system for the reasons stated above.

Of particular interest to us are stealthy attackers [11, 12]. They continue normal operation of the compromised tasks while gathering timing information about other, critical tasks. The knowledge about the timing of critical tasks allows to determine the point in time when a directed attack is most effective, e.g., immediately before a safety-critical victim task is run. The timing information required for coordinating such a stealthy attack can be inferred via various means which we will detail further below.

While there exist mitigation strategies for closing side channels in real-time systems [52, 69], these methods are incomplete as they do not consider all possible side channels. Additionally, systematically closing side channels typically entails significant performance overheads, e.g. when flushing caches prior to scheduling a lower classified task [33]. Meltdown [45] and Spectre [37] are recent examples demonstrating the difficulty of identifying and closing such channels in sufficiently complex architectures. Exploiting non-architectural channels (e.g., cache allocation) as communication medium, Meltdown and Spectre extract confidential information from speculative processor state, breaking security on most Intel and many ARM and AMD processors.

We further assume the real-time system features isolation mechanisms for enforcing and protecting the schedule of tasks and for limiting direct access to the memory of other tasks. Real-time operating systems (RTOSs) that feature memory isolation support this assumption unless attackers are able to penetrate the operating system. For the scope of this thesis, we exclude the possibility of OS penetration. One immediate consequence of this isolation assumption is that when attackers infiltrate the system, they are inherently constrained by properties of the system and its architecture for subsequent attacks on safety-critical tasks. Time-triggered systems isolate tasks in the time domain, which prevents influencing other tasks by manipulating the execution time of a compromised task. That is, in contrast to event-triggered scheduling, each task is confined to its execution window and thus the actual task execution time has no influence on subsequent tasks. Time-triggered systems

therefore provide temporal isolation of CPU time irrespective of the actual execution time of tasks.

The assumed impenetrability of the RTOS ensures that attackers can neither directly influence the scheduler nor gain direct access to offline constructed schedules. Instead, the attacker has to infer the current schedule from observations about the system behaviour. As we show in Chapter 7.4.1, schedules typically carry too little information to remain secure over extended periods of time even if this information is leaked over channels with low bandwidth. Furthermore, we assume that the global time base remains under exclusive control of the operating system and that it cannot be affected by the attacker.

Time-triggered systems eliminate CPU time as shared resource over which information can be leaked and through which other tasks may be influenced. However, other resources remain through which attackers may gain information and through which they can impact the timing behaviour of other tasks. Attackers may purchase or reverse engineer the system to figure out its timing information, i.e. the schedule of its tasks. Another option is to construct side channels using shared resources, or employ covert timing channels, such as the scheduling-covert-channel described by Boucher et al. [10]. We therefore assume that attackers can infer timing information.

One prominent example of such a shared resource is the processor cache. We detail how an attacker is able to use inferred timing information below. Uncompromised tasks leave the cache behind in a predictable state, but compromised tasks can put it into a state that may not have been anticipated when computing the worst-case execution time of subsequent tasks. Access controls and partitioning techniques like cache colouring [44] or bank colouring [75] may constrain the attacker. However, each such countermeasure negatively impacts system performance and they are generally not complete. For example, Bechtel et al. [9] demonstrate a Denial-of-Service attack in caches that are not prepared for partitioning and that therefore retain shared resources (e.g., for cache-miss handling). Once exhausted by the attacker's accomplice task, these resources are no longer available to the victim, stalling its execution until all outstanding write-backs are handled. The authors only consider the accomplice and victim task to be present, which are both running on different cores and - except for sharing the cache - are isolated from each other. However, usually there are multiple tasks running in a real-time system and the attacker may not hit the victim in all cases. If the attacker has inferred the schedule instead, they are able to discern the best point in time to attack. This allows the attacker to remain stealthy until the time has come when the victim is most vulnerable and to better evade attack detection mechanisms.

In summary, we assume an attacker who has infiltrated non-critical tasks of a time-triggered single core system and has inferred its timing information (i.e. the schedule) in order to mount a directed attack against a critical victim task. We show how to mitigate this type of attack in the next chapter.

5 SINGLE CORE MITIGATION STRATEGIES

The attacker's goal is to predict when a victim task is scheduled. Especially a point in time where the victim is scheduled immediately after a compromised task is of special interest to the attacker as this scenario facilitates an attack directly aimed at the victim. Our mitigation strategy is therefore to impede predictions about the points in time when the victim executes. While we do not prevent timing inference, i.e. we assume the attacker may gain information about the scheduling parameters, we are able to counter predictions by changing the points in time when tasks are executed at runtime. For this purpose, we present two strategies to mitigate directed attacks in this chapter. The first mitigation strategy takes task set parameters as input and randomizes the schedule online at job level while maintaining timeliness constraints. The second mitigation strategy consists of a set of offline constructed schedules one of which is randomly chosen at the end of each hyperperiod during runtime. We assume the randomized values for the mitigation strategies stem from a trusted component and are sufficiently unpredictable for our purpose.

5.1 Online Slot-Level Randomization

This mitigation strategy impedes the ability of an attacker to make predictions by randomizing job executions in a time-triggered system at runtime. Schedules for time-triggered systems are typically constructed offline [22], where real-time constraints are resolved and represented in a scheduling table. For each job in the system, its execution is mapped to one or more slots of the time-triggered system. Online randomization may violate timeliness constraints, e.g., the deadline. Therefore, our approach analyses the scheduling parameters offline and maps timing constraints of jobs onto execution windows. Execution windows are time intervals defined by the earliest start time of a job and its deadline. Each job has to start and finish execution within its execution window. Modification of execution windows solves precedence constraints as execution windows are able to force one task to execute after the other. Additionally, if one of the goals of the system is to achieve low jitter, we can reduce the size of execution windows accordingly.

During runtime, we randomize job execution within their respective execution windows. While we confine jobs to their execution windows, they still share the same processor so we also have to guarantee that their execution does not lead to a deadline miss of other jobs. Slot shifting is a scheduling algorithm which introduces the concept of spare capacities to ensure timely execution [27]. We adopt this concept to guarantee task execution within their respective execution windows even though the scheduling decision is randomized.

5.1.1 Adaptation of Slot Shifting

The slot shifting algorithm was introduced to allow for flexibility in static time-triggered schedules. If enough CPU resources are available to guarantee execution, it allows aperiodic tasks at runtime. In order to keep track of resource availability, slot shifting introduces the concept of spare capacity and capacity intervals for bookkeeping. We adopt this concept to ensure our randomization of job execution does not lead to violations of timeliness constraints.

Slot shifting uses a discrete time model [38], where the time interval which separates two successive events (i.e. the granularity of the system) is called a slot [61]. We analyse the time-triggered schedule and its task set offline to determine available leeway and unused resources in the schedule for subsequent online adjustment. In order to track the available leeway of jobs in each execution window, a capacity interval is created for each distinct deadline in the system. Jobs with the same deadline belong to the same capacity interval. Capacity intervals must not overlap each other. Therefore, we define the start of a capacity interval I_j, $start(I_j)$, as the maximum of the earliest start time $est(I_j)$ of jobs τ_i in this interval and of the end of the previous, i.e. preceding, capacity interval:

$$start(I_j) = \max(end(I_{j-1}), est(I_j)), \text{ where } est(I_j) = \min(est(\tau_i)) \; \forall \tau_i \in I_j \quad (5.1)$$

The end of the capacity interval is determined by the common deadline of all $\tau_i \in I_j$. If needed, empty capacity intervals without assigned jobs are created to fill gaps between capacity intervals with assigned jobs. The next sections show how to create capacity intervals, how to calculate spare capacity and update throughout job execution, and how the scheduler chooses the next job to execute.

5.1.2 Example of Capacity Interval Creation

Figure 5.1 shows an example job set with earliest start times est_i, worst case execution times C_i and deadlines d_i. Three distinct deadlines exist for that job set, thus at least three capacity intervals have to be created. The first interval I_1 starts at 0, which is the earliest release time of its assigned set of jobs $\{\tau_1\}$, and ends at the deadline of its jobs, which is 4. The job assigned to the next interval, τ_2, shares the earliest start time of τ_1, but according to Equation 5.1, a capacity interval is not allowed to start before the end of the previous interval. Note that capacity intervals do not overlap, while execution windows may. Thus, I_2 starts at 4 and ends at the deadline of its assigned set of jobs $\{\tau_2\}$, which is 7. We create interval I_3 accordingly. We show the resulting capacity intervals in Figure 5.1.

τ_i	est_i	d_i	C_i
τ_1	0	4	2
τ_2	0	7	1
τ_3	4	8	2

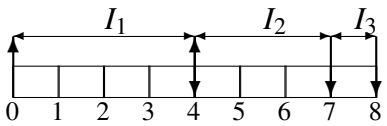


Figure 5.1: Job set and corresponding capacity intervals

In the next section, we explain the calculation of initial spare capacities and how they are updated after job execution.

5.1.3 Spare Capacity Calculation and Update

Spare capacities are a means to track "free" slots in a capacity interval. The spare capacity $sc(I_j)$ of a capacity interval I_j is equal to the amount of free slots in I_j. We define $sc(I_j)$ as the interval length minus the sum of worst case execution times C_i of all its jobs τ_i minus slots borrowed from the succeeding interval (denoted as negative spare capacity). Equation 5.2 below shows its calculation:

$$sc(I_j) = |I_j| - \sum_{\tau_i \in I_j} C_i + \min\left(sc(I_{j+1}), 0\right) \tag{5.2}$$

We calculate spare capacities starting from the last capacity interval in the hyperperiod to the earliest. Borrowing is a mechanism which occurs in those cases where the current capacity interval provides insufficient slots to accommodate all its jobs, which results in a negative spare capacity (e.g. I_3 in Figure 5.2). Capacity intervals with a negative spare capacity borrow the needed amount of slots from the preceding interval. Negative spare capacities do not necessarily imply infeasibility in the scheduling sense.

If the task set is schedulable, the first capacity interval has a non-negative spare capacity after we have calculated all spare capacities. Positive spare capacities represent the amount of unused resources and leeway [27] of an interval which can be given to other tasks with overlapping execution windows to adjust the schedule. Such adjustments may require updating spare capacities. At runtime, we update the spare capacities after each slot to reflect the impact of scheduling decisions on the availability of "free" slots.

We consider three different cases for spare capacity updates:

1. No job executes in a given slot, i.e. the processor idles. In this case we have to decrease the spare capacity of the current capacity interval by one.

2. A job executes which belongs to the current capacity interval. In this case the spare capacity of the current interval does not change because the WCET of this job is already considered in its calculation.

3. A job executes which belongs to a later capacity interval. In this case the current interval's spare capacity needs to be decreased by one. Executing the job ahead of time also frees resources in its assigned interval. We can therefore increase the spare capacity of the job's interval by one. If this capacity increase happened on a negative spare capacity (i.e., the job's interval is borrowing from another capacity interval), we also increase the spare capacity from the interval from which it borrows, as it needs to lend one slot less. Cascaded borrows are resolved recursively in the same fashion.

5.1.4 Online Job Randomization

After introducing the concepts of spare capacities and capacity intervals, we explain how the scheduler selects tasks from the ready queue. At the beginning of each slot, we invoke the online scheduler to select the next job from all tasks in the ready queue at random. We consider the idle task to be part of the ready queue in order to allow for more permutations of the schedule. We assume that the randomization function aims for an equal distribution of jobs over the available slots considering multiple hyperperiods. Each job thus has the same probability to get chosen for execution in a certain slot within its execution window to avoid predictions.

Even though we select tasks randomly, we have to guarantee that no scheduled job violates the deadline constraints of other jobs. Thus, before taking a scheduling decision, we check if the spare capacity of the current capacity interval is greater than zero. If this condition is fulfilled, any job is allowed to run, as sufficient time remains in the current and later intervals such that no job misses its deadline. In other words, as long as the schedule has leeway, each job in the ready queue may be selected to execute in a certain slot. Otherwise, if the spare capacity of the current interval drops to zero, there is no more leeway to schedule arbitrary jobs. However, because we have already considered jobs of the current capacity interval in the spare capacity computation and because all such jobs share the same deadline, we can still randomize their execution. That is, in the case of zero leeway, the online scheduler randomly selects among the jobs belonging to the current capacity interval. After the job has run during its assigned slot, we update spare capacities as shown in Section 5.1.3. In the next section, we show an example of spare capacity calculation, spare capacity update, and job selection.

5.1.5 Online Slot-Level Randomization Example

Let us illustrate the proposed scheduling algorithm for the example jobset depicted in Figure 5.1. First, we have to calculate the initial spare capacities of the capacity intervals using Equation 5.2. Starting at the last capacity interval, I_3, its spare capacity is the difference between the interval length of 1 and the worst case execution time of its assigned jobs, τ_3, which results in a negative spare capacity of -1. I_2 has an interval length of 3, from which we subtract the worst case execution time of τ_2 (i.e., $C_2 = 1$) and the slots borrowed by the preceding interval I_3 (by adding $sc(I_3) = -1$), which results in a spare capacity of 1. We calculate the spare capacity of I_1 accordingly. Figure 5.2 shows the resulting initial spare capacities in the column for time $t = 0$.

time t	0	1	2	3	4	5	6	7	8
$sc(I_1)$	2	2	1	0	0	0	0	0	0
$sc(I_2)$	1	1	1	2	2	1	1	0	0
$sc(I_3)$	-1	-1	-1	-1	-1	-1	0	0	0

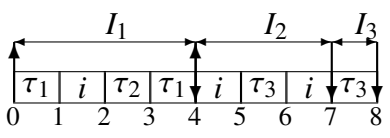


Figure 5.2: Left: spare capacities of I_1, I_2 and I_3 over time, Right: randomized schedule

At time $t = 0$, the scheduler sees that the spare capacity of the current interval I_1 is positive. It picks τ_1 randomly for the first slot at $t = 0$ from the list of ready jobs τ_1, τ_2, and the idle job i. As τ_1 executes within its own interval, the current spare capacity does not change and remains positive. The idle job i is selected to execute during the next slot starting at $t = 1$, necessitating a decrease of the spare capacity by one. τ_2 is randomly selected for time $t = 2$. τ_2 does not execute within its own capacity interval, therefore we reduce $sc(I_1)$ by one and increase $sc(I_2)$ by one, since τ_2 belongs to interval I_2 and I_2 does not borrow from I_1. $sc(I_1) = 0$ at $t = 3$ constrains the online scheduler to select from the set of jobs $\{\tau_1\}$ that belong to I_1. τ_3 becomes active at time $t = 4$ and the scheduler selects it to execute at time $t = 5$ after picking the idle thread to run at $t = 4$. This is valid, as $sc(I_2)$ is positive, and thus we reduce $sc(I_2)$ by one and increase the capacity interval of τ_3, I_3, by one. However, at this time, I_3 is still borrowing one slot from I_2. τ_3 executed prior to its own capacity interval, thus I_3 needs to borrow one slot less from I_2 and therefore we increase $sc(I_2)$ by one, resulting in no change of $sc(I_2)$. In summary, $sc(I_2)$ stays at 1 and $sc(I_3)$ is increased by one. We show further exemplary scheduling decisions and the resulting spare capacity updates in Figure 5.2. The schedule does not represent a hyperperiod as this is not necessary for illustratory purposes. The algorithm operates the same whether considering the hyperperiod or not.

5.1.6 Online Slot-Level Randomization Conclusion

The original slot shifting algorithm in [27] and [61] integrates aperiodic tasks into a time-triggered schedule. We adopt the concept of capacity intervals and spare capacities to guarantee timely execution of periodic jobs within their execution windows without violating constraints of other jobs. Thus, our algorithm needs to create data structures for the capacity intervals and their respective spare capacities for bookkeeping. We update the spare capacities at runtime to keep track of scheduling decisions.

Combining time-triggered scheduling with our online slot-level randomization impedes predictions about the schedule. Since the scheduler randomly selects the next job at runtime, predictions about which job runs next are not possible as long as execution windows allow for leeway. Furthermore, time-triggered scheduling inherently confines application-level leakage to shared resources which are held across slots [68]. An investigation of leakage countermeasures for such resources is out of the scope of this thesis. While our randomization algorithm does not allow for slot-level determinism typical for time-triggered systems, it still allows for execution window determinism [28]. The following section introduces the second mitigation strategy, offline schedule-diversification.

5.2 Offline Schedule-Diversification

This mitigation strategy computes multiple schedules offline and switches between them at hyperperiod boundaries during runtime. Time-triggered systems typically follow a schedule whose constraints, e.g. deadline and precedence constraints, have been resolved offline and which repeats itself during runtime. Resolving scheduling constraints offline greatly reduces runtime scheduling complexity and ensures lower runtime overheads. However, it also increases the chance of attackers to guess the schedule and launch directed attacks. For example, repeating the same offline computed schedule over and over allows an attacker to infer the schedule, as illustrated in [52]. Attackers can use the information gained to coordinate directed attacks from compromised tasks scheduled later in the same hyperperiod or in subsequent hyperperiods. To partially mitigate this threat vector, we randomly switch schedules at the end of each hyperperiod. As a consequence, even when attackers are able to recognize different schedules and have enough memory available to store them, the more schedules have been generated, the harder it is for the attackers to recognize which schedule has been chosen for the current hyperperiod and less time remains to perform a directed attack. In particular, if the attackers are not able to identify the current schedule in time for their attack, they miss the opportunity to perform a directed attack.

We show in Chapter 7.5 that computing and storing all possible, feasible schedules in memory is impractical. However, in non-embedded systems we foresee the continuing generation of schedules in a non real-time subsystem (e.g., in a sufficiently protected external control station) which updates the set of schedules in the real-time system. This way, once a new set of schedules has been produced, e.g., by recombining precomputed and stored schedules, the real-time system can switch to the new set at the end of the hyperperiod. Double buffering, signing and encryption of schedules ensures that the current set of schedules remains valid while the system validates the confidentiality and integrity of the new schedules (e.g., in a background task). Irrespective of update possibilities, the selected subset of schedules out of the set of all feasible schedules for a given task set should impede directed attacks. We present two criteria to select subsets that complicate directed attacks in addition to guaranteeing deadlines and respecting task precedence constraints.

5.2.1 Random Schedule Set Selection

For the sake of low implementation complexity, we can randomly select the subset. That is, we create schedules at random and check if they meet all scheduling constraints. The schedules fulfilling this requirement form the set of schedules for the system. We stop schedule creation after we have constructed a certain number of feasible schedules. We recommend this method for large subsets, when enough memory is available to store a large number of different schedules. If the subset is large enough, the random selection process provides a set of schedules with a schedule entropy close to the set of all feasible schedules. Other criteria impose more constraints on the selection process and therefore increase its complexity.

5.2.2 Schedule Entropy

Another criterium for schedule selection is schedule entropy as presented in [74], which uses an approximation of the Shannon entropy. Using a metric for entropy, we are able to quantify the diversity of a set of schedules, which means we are able to compute the set of schedules that maximizes its diversity. This allows us to randomly pick a schedule in this set and give the attacker the least possible amount of information about which task executes in which slot. Ideally, an attacker eavesdropping to the schedule through a side-channel would see the task set execution to be as diverse as possible.

In [43], we investigate the fundamental limitations to the achievable diversity imposed by the task set characteristics. There, we show how to determine the maximum upper-approximated entropy for a given task set and a given set of schedules. We acknowledge that any constraint on the task set, whether deadline, period or precedence constraint, limits the maximum amount of diversity that can be achieved. From this set of schedules, we show how to compute the minimum number of schedules needed to achieve the maximum schedule diversity.

While schedule entropy is a measure for the diversity of schedules, it does not quantify security per se. If a set of schedules is more diverse than another set of schedules, the schedules have less similarities. Thus, the more diverse the schedules, the lower the probability that a job is executed in a the same slot over several schedules. This gives an attacker less information about the job execution itself, yet makes sure the schedules are easier to distinguish from one another. Depending on the memory and side channel bandwidth available to the attacker, a small set of diverse schedules might be easy to figure out and store. Then, if the schedules are maximally diverse from each other, the schedule chosen online might be easy to recognize during execution and the attacker might execute a directed attack nonetheless. We discuss this possible disadvantage of using schedule entropy as metric in more detail in Chapter 7.6.

6 EXTENSION TO MULTICORE SYSTEMS

Previous chapters considered single core real-time systems and introduced novel attack vectors as well as mitigation strategies for these type of systems. This chapter extends the assumptions to multicore systems and presents novel attack vectors and mitigation strategies tailored to multicore systems. First, we present our system model for multicore systems. From the system model we derive the threat model aimed at multicore time-triggered systems and introduce novel attack vectors. We then propose mitigation strategies for these attack vectors. In order to let this chapter stand on its own, some overlap with a previous chapter on single core models exists.

6.1 Multicore System Model

We assume a time-triggered real-time system implemented on a multicore platform, following a partitioned schedule. A real-time operating system (RTOS) is present, that is, we do not consider bare metal real-time implementations. The real-time system runs safety-critical as well as non-safety-critical tasks. For each core, a local schedule is constructed offline and then adjusted online according to task set parameters. Section 6.4 provides more detail on the online adjustments. We assume the schedule is correct and that the task set is schedulable. Precautions such as authenticated boot are in place to ensure that the schedule is correctly deployed in the real-time system. Safety-critical tasks are replicated and replicas are distributed across cores to benefit from improved resilience in case one core fails. The implementation of these replicas follows an implementation diversification approach to increase system dependability. We assume the system is able to tolerate up to f faults of arbitrary kind simultaneously. Arbitrary faults include accidental faults, malicious faults, crash faults as well as Byzantine faults. The replication degree, i.e. the number n of replicas required to achieve the mentioned fault tolerance, depends on the system's fault model, its achieved synchrony and on the kind of agreement, e.g. single value versus interval mid point. Our approach is prepared for any combination of the above.

Time-triggered systems and their schedulers rely on a global time base. The scheduler on each core has access to the global time base. We consider the global time base protected from attacks, as it is a critical system component. Global time is divided into slots of the same size, which are the granule for job preemption. Our focus lies on CPU-level scheduling, hence we do not consider time-triggered networks or communication channels. Moreover, we assume a time-triggered system without asynchronous event activation.

For each core c, we are given a task set $\mathcal{T}^c = \{\tau_1^c, \tau_2^c, \ldots, \tau_n^c\}$ of n^c periodic tasks. Each task τ_i^c is defined as the tuple $\tau_i^c = \{C_i^c, T_i^c, D_i^c\}$, where C_i^c is the worst-case execution time of the task, T_i^c is the task's period, and D_i^c is the task's relative deadline. We denote the absolute deadlines of tasks, i.e. their jobs, with d_i^c. To guarantee the correct execution of the task set, time-triggered systems ensure that a time equal to or greater than the worst-case execution time is assigned to each job of each task. This means that the time budget is allocated to the task even if the job completes its execution early. Throughout the rest of the work, we assume that $D_i^c = T_i^c$, adopting the implicit deadline task model. We define the time interval between task release and task deadline as this task's execution window.

6.2 Multicore Threat Model

So far, we have assumed that attackers are limited to non-critical tasks. Through the use of replication, we can also protect against stronger adversaries which are capable of compromising up to f replicas of an n-fold replicated critical subsystem. However, we need to ensure they never cross the threshold of compromising more than f replicas. We assume attackers are able to successfully infiltrate the system through undetected vulnerabilities and compromise non-safety-critical tasks. In particular, we assume that safety-critical tasks are sufficiently shielded against direct attacks which requires attackers to find a pathway through non-safety-critical tasks. Less stringent evaluation requirements make non-safety-critical tasks, including non real-time tasks, primary targets for attackers. Firewalls and gateways protecting safety-critical tasks in autonomous vehicles and planes support this assumption. Even though we assume intrusion detection [19, 48, 51, 76], hardening mechanisms and other defenses against common attack vectors (e.g. DoS attacks) are in place, we acknowledge that these techniques are imperfect. The past has shown attackers to find complex pathways into systems nonetheless and compromises may go undetected. In our case, we restrict the compromise to non-safety-critical parts of the system for the reasons stated above.

Stealthy attackers are of particular concern to us [11, 12]. They continue normal operation of the compromised tasks while gathering timing information about other, critical tasks. The knowledge about the timing of critical tasks allows to determine the point in time when a directed attack is most effective, e.g., immediately before a safety-critical victim task is run. The timing information required for coordinating such a stealthy attack can be inferred via various means which we detail later in this chapter.

While there exist mitigation strategies for closing side channels in real-time systems [52, 69], these methods are incomplete as they do not consider all possible side channels. Additionally, systematically closing side channels typically entails significant performance overheads, e.g. when flushing caches prior to scheduling a lower classified task [33]. Meltdown [45] and Spectre [37] are recent examples demonstrating the difficulty of identifying and closing such channels in sufficiently complex architectures. Exploiting non-architectural channels (e.g., cache allocation) as communication medium, Meltdown and Spectre extract confidential information from speculative processor state, breaking security on most Intel and many ARM and AMD processors.

We further assume the real-time system features isolation mechanisms for enforcing and protecting the schedule of tasks and for limiting direct access to the memory of other tasks. Real-time operating systems (RTOSs) that feature memory isolation support this assumption unless attackers are able to penetrate the operating system. For the purpose of this thesis, we exclude the possibility of OS penetration. One immediate consequence of this isolation assumption is that when attackers infiltrate the system, they are inherently constrained by properties of the system and its architecture for subsequent attacks on safety-critical tasks. Time-triggered systems isolate tasks in the time domain, which prevents influencing other tasks by manipulating the execution time of a compromised task. That is, in contrast to event-triggered scheduling, each task is confined to its execution window and thus the actual task execution time has no influence on subsequent tasks. Time-triggered systems therefore provide temporal isolation of CPU time irrespective of the actual execution time of tasks.

The assumed impenetrability of the RTOS ensures that attackers can neither directly influence the scheduler nor gain direct access to offline constructed schedules. Instead, the attacker has to infer the current schedule from observations about the system behaviour. As we show in Section 7.4.1, schedules typically carry too little information to remain secure over extended periods of time even if this information is leaked only over low bandwidth channels. Furthermore, we assume that the global time base remains under exclusive control of the operating system and that it cannot be affected by the attacker.

Time-triggered systems eliminate CPU time as shared resource over which information can be leaked and through which other tasks may be influenced. However, other resources remain through which attackers may gain information and through which they can impact the timing behaviour of other tasks. Attackers may buy or reverse engineer the system to figure out the timing information, i.e. the schedule of its tasks. Another option is to construct side channels using shared resources, or employ covert timing channels, such as the scheduling-covert-channel described by Boucher et al. [10]. We therefore assume that attackers can infer timing information.

One prominent example of such a shared resource is the processor cache. Below we detail how an attacker is able to use inferred timing information. Uncompromised tasks leave the cache behind in a predictable state, but compromised tasks can put it into a state that may not have been anticipated when computing the worst-case execution time of subsequent tasks. Access controls and partitioning techniques like cache colouring [44] or bank colouring [75] may constrain the attacker. However, each such countermeasure negatively impacts system performance and they are generally not complete. For example Bechtel et al. [9] demonstrate a Denial-of-Service attack in caches that are not prepared for partitioning and that therefore retain shared resources (e.g., for cache-miss handling). Once exhausted by the attacker's accomplice task, these resources are no longer available to the victim, stalling its execution until all outstanding write-backs are handled. In this system, only the accomplice and victim task are present, both running on different cores and, except sharing the cache, are isolated from each other. However, usually there are multiple tasks running in a real-time system and the attacker may not hit the victim in all cases. If the attacker has inferred the schedule instead, they are able to discern the best

point in time to attack. The attacker can remain stealthy until the time of attack has come when the victim is most vulnerable.

In summary, we assume an attacker who has infiltrated tasks of a multicore system and has inferred its timing information (i.e. the system schedule) in order to mount a directed attack against critical victim tasks. The attacker needs to compromise more than f replicas of the safety-critical task in order for the attack to succeed. In the next section we propose three different attack vectors based on this assumption.

6.3 Multicore Attack Vectors

In the previous section we detailed how attackers operate and try to mount a directed attack. This leaves a crucial question: which are the points in time where a directed attack on a multicore system is most successful? In order to answer this question we considered the points in time when the attacker has the most influence on other tasks: the hijacked task runs directly before or in parallel to the victim task. We identified three attack vectors (AV) which consider this temporal proximity and depict them below with corresponding figures:

AV1: A compromised task or replica attacks a victim task executing next on the same core.

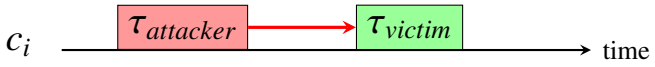


Figure 6.1: Attacker and victim task on the same core c_i

Attack vector 1, as Figure 6.1 illustrates, equals an attack on a single core system. An example for AV1 is a compromised task that deviates from its analysed behaviour to create the worst possible cache state for the subsequent task, e.g. requiring write backs throughout the cache hierarchy for all lines to be allocated. The effect of such an attack is the strongest for victims scheduled immediately after attackers. If intermittently scheduled tasks are present which preempt the victim task, they reduce the attacker's influence. They possibly fetch some of the state required by the victim task during execution and write some malicious cache content back to memory.

AV2: A compromised task or replica attacks a task or replica executed in parallel on another core.

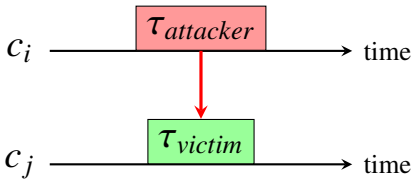


Figure 6.2: Attacker on core c_i attacks victim task running in parallel on another core c_j

Figure 6.2 portrays the second attack vector, AV2. Here the attacker attacks the victim task running in parallel on another core, possibly influencing its victim over

shared resources. An example for AV2 is to replace cache lines before they are requested, for example, turning cache lines from shared to a remotely modified state. Attacks along this vector are most effective when executed in parallel to the victim task and when at least some state is shared between the attacker and victim task, e.g. from a replica sharing code and read-only data.

AV3: A combination of both attack vectors AV1 and AV2, i.e., attacks are mounted from one core shortly before a task or replica is run on another core.

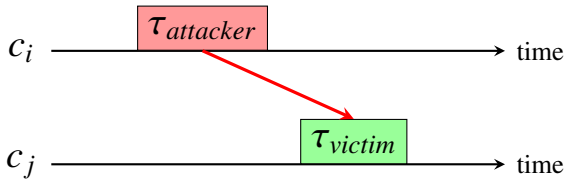


Figure 6.3: Attacker on core c_i attacks victim task running next on another core c_j

An attack along the combined attack vector, AV3, aims at creating the effect of AV1 through the means of AV2. As Figure 6.3 illustrates, an attacker tries to influence a victim task who is going to execute soon on another core. The attacker could, e.g., try to evict all shared state from the caches at the victim's core.

For all three attack vectors, we assume the attacker has knowledge of the deterministic time-triggered schedule on its own and other cores. As long as the attacker compromises only a minority of replicas, the replicated system is able to tolerate the attack. However, if the determinism inherent to time-triggered systems allows attacks to spread to replicas on other cores, this assumption does not hold anymore. To prevent this directed attacks from succeeding, we present our mitigation strategies in the next section.

6.4 Multicore Mitigation Strategies

Previously introduced mitigation strategies only regard single core systems. In this section, we extend our mitigation strategies to multicore systems. The previous section shows the additional attack surface multicore systems face, but also points out the additional resilience gained from replication. Since all three attack vectors rely on the knowledge of the points of execution of the victim tasks, we build upon our previously introduced mitigation strategies for single core systems to extend them to multicore systems.

6.4.1 Online Slot-Level Randomization

The first mitigation strategy is online slot-level randomization as presented in Section 5.1. Slot-level randomization runs locally on each core and prevents predictions about the points in time when tasks, including replicas, are executed, on the same core as well as on other cores. Attackers are unable to predict replica execution and cannot rely on deterministic assumptions to coordinate their attack. This strategy prevents the directed attack we consider for all attack vectors and thus mitigates AV1, AV2, and AV3.

Depending on the randomization method, it is also possible to synchronize random task execution over cores. For example, if all cores use the same seed to initialize their random number generator at the same time, their (pseudo-)random choices are the same. This property is a trade-off between reliability and security. While synchronization might be desirable for redundancy purposes among cores, e.g. if some serve as redundant copies for others, unless the task implementations have been diversified, an attacker might be able to attack a majority of replicas at the same time. Therefore, we do not recommend this approach unless there are strong reasons for synchronization. Moreover, when the random number generators are not synchronized, tasks still adhere to their execution window and thus synchronization is possible with a more coarse granule.

6.4.2 Offline Schedule-Diversification

The second mitigation strategy employs offline schedule creation which possesses the advantage to solve complex constraints before runtime. Offline schedulers create schedules and solve all scheduling constraints before system deployment. In addition to the deadline constraints already discussed in Section 5.2 (and possibly further constraints to accommodate for precedence or jitter), we impose an additional constraint: at the end of each slot, no two cores may switch to the same job. This way, schedules across cores are diverse and we mitigate AV2, by avoiding replica execution at the same time.

However, the offline constructed schedule still provides the determinism an attacker can exploit for AV1 and AV3. As the constructed schedules are diverse, each core's RTOS stores all schedules. One option is that cores switch to another schedule at the end of the hyperperiod in an offline defined, deterministic way, such that each core executes a different schedule. This nonetheless provides deterministic execution to the attacker, but the window of repetition, that is, the size of the hyperperiod is larger.

Alternatively, the scheduler may choose the next schedule at the end of the hyperperiod at random. If the set of diverse schedules is diverse and big enough, the possibility of retrieving information using the schedule is low. Cores still may randomly execute the same schedule at the same time and thus replicas may be executed in parallel, but the attacker is not able to rely on this possibility in a deterministic fashion. This approach prevents directed attacks.

6.4.3 Combined Approach

In this approach, we combine both mitigation strategies, online slot-level randomization and offline schedule-diversification. The execution of safety-critical task replicas is of particular interest to us as we want to prevent an attacker from compromising more than f replicas at the same time. First, we create non-overlapping execution windows of replicas offline, i.e., we restrict replica execution windows to not overlap each other to fulfill this constraint. As consequence, replicas have different earliest start times and deadlines on each core. This constraint eliminates AV2, where replicas are scheduled at the same time. Then, during runtime, we use online slot-level randomization to prevent predictions and thus eliminate attack vectors AV1 and AV3 enabled by a deterministic schedule. Compared to the pure online approach presented in Section 6.4.1, the combined strategy never schedules replicas in parallel. However, depending on the task set, the restriction of replica execution windows may leave few leeway in the schedule for the online randomization algorithm.

7 EVALUATION

In this chapter, we evaluate our mitigation strategies with a real-world case study, ROSACE (Research Open-Source Avionics and Control Engineering) [56], and compare them against static time-triggered scheduling. The ROSACE case study provides task set parameters for a safety-critical real-time system of an aircraft: the longitudinal flight controller. First, we approximate the runtime scheduling overhead using the task set parameters of this case study and measurements from previous work on a cycle-accurate ARM simulator. Then we implement the case study, our mitigation strategies and time-triggered scheduling on a real-time operating system, FreeRTOS, and provide details of both the implementation and measurement methodology. We provide an analysis of scheduling overhead and memory cost of all three scheduling algorithms, calculate the achievable solution space for our mitigation strategies and the probe effect, and discuss determinism granularity and schedule diversity. Our results show that both mitigation strategies provide feasible alternatives compared to static time-triggered scheduling.

7.1 ROSACE Case Study

ROSACE is a practical, real-world example of a real-time system in a safety-critical avionics environment. Pagetti et al. [56] carried out a case study of a longitudinal flight controller of an aircraft. The longitudinal flight controller helps the pilot to accurately track altitude, vertical speed and airspeed of the aircraft. Pagetti et al. describe two control loops: the *Va_control* loop handles airspeed control by maintaining the desired airspeed Va; the second control loop – altitude control – combines *altitude_hold* and *Vz_control*. First, *altitude_hold* translates altitude commands to vertical speed commands. Then, *Vz_control* tracks the vertical speed Vz of the aircraft. Both control loops are fed with filtered data: h, az and q for altitude, vertical acceleration and pitch rate, respectively. Vertical airspeed Vz and true airspeed Va are also inputs to the control loops. We show the design of the controller in Figure 7.1.

According to Pagetti et al. [56], the closed-loop system with continuous-time controllers can tolerate delays of up to roughly 1 second before destabilizing. To preserve stability as well as to increase performance, Pagetti et al. chose lower sampling periods of 50 Hz for the digitalization tasks of the three controller blocks and 100 Hz for the filter tasks which feed the data to the controller. Pagetti et al. derived worst case execution times of all tasks using a measurement-based approach by measuring the repeated execution of a task in isolation. The granularity the authors chose for the measuring clock was 100 μs,

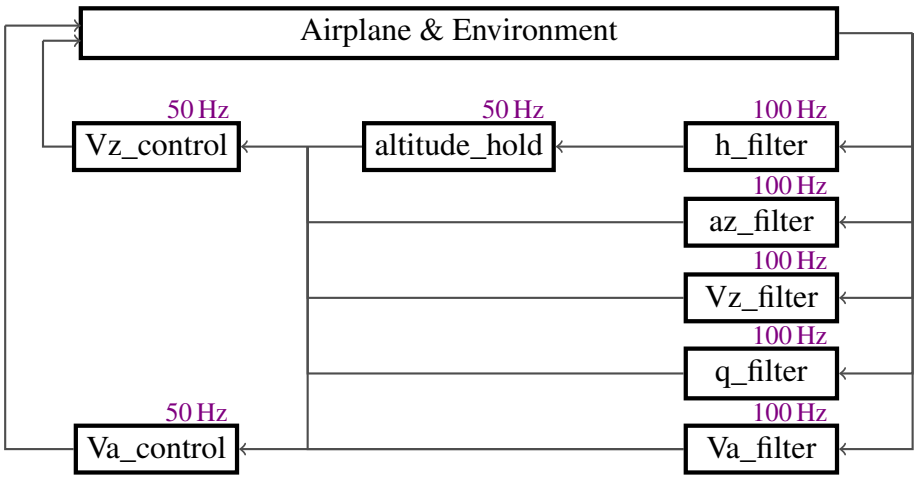


Figure 7.1: Longitudinal flight controller design

Taskname	Frequency	WCET
Vz_control	50 Hz	100 μs
Va_control	50 Hz	100 μs
altitude_hold	50 Hz	100 μs
h_filter	100 Hz	100 μs
az_filter	100 Hz	100 μs
Vz_filter	100 Hz	100 μs
q_filter	100 Hz	100 μs
Va_filter	100 Hz	100 μs

Table 7.1: Flight controller task set[56]

thus the worst case execution times for the tasks shown are the same as they presumably finished execution in that granule. Table 7.1 shows the task set with implicit deadlines for the longitudinal flight controller. In this work, we do not consider environment simulation tasks as they are not part of the controller but only of the test environment.

7.2 Runtime Overhead Approximations using MPARM Measurements

In this and the following sections, we first approximate, then measure the runtime overhead of the mitigation strategies presented earlier. We base the runtime overhead approximations on previous work that carried out exhaustive measurements. Both mitigation strategies share most of the basic functions of Schorr's [61] slot shifting algorithm. Schorr measured the runtime overhead of the unmodified slot shifting algorithm on a cycle-accurate ARM quadcore simulator – MPARM – with ARM7 cores running at 200 MHz, 8kB 4-way set associative L1 cache, 8kB direct mapped L1 instruction cache, 1MB core-private

memory and 1MB shared memory. Based on Schorr's measurements, we approximate runtime overhead for both mitigation strategies. In a later section, we show how close these approximations are when we compare them to measurements of an implementation of our mitigation strategies on FreeRTOS.

7.2.1 Approximations for Online Slot-Level Randomization

We adapt the ROSACE case study for the MPARM simulator, i.e. we construct the execution windows of all tasks from the task set presented in Table 7.1. Schorr [61] suggests 200,000 clock cycles as slot shifting slot length. The processor cores in ROSACE run at 1.2GHz, which results in 167 μs for 200,000 clock cycles. We choose 200 μs as slot length to evenly divide the task periods into slots. Task execution is non-preemptive, as the worst case execution times are smaller than the slot length and were derived running the tasks in isolation. Table 7.2 shows the resulting execution windows.

Name	**Start**	**End**	**WCET**
h_filter	0	50	1
az_filter	0	50	1
Vz_filter	0	50	1
q_filter	0	50	1
Va_filter	0	50	1
h_filter	50	100	1
az_filter	50	100	1
Vz_filter	50	100	1
q_filter	50	100	1
Va_filter	50	100	1
altitude_hold	0	100	1
Vz_control	0	100	1
Va_control	0	100	1

Table 7.2: Execution windows in terms of slots for MPARM

Schorr provides minimum and maximum runtimes of all parts of the slot shifting algorithm for single core execution. Using the timing measurements of [61], shown in Table 7.3, we approximate the runtime overhead of slot-level randomization, when executed on the same processor.

Slot-level randomization invokes the same functions to update spare capacities and the ready list. The cost of the function to update spare capacities increases with the number of intervals due to cascaded borrowing in the worst case. However, according to the slot shifting algorithm as explained in Section 5.1.1, only 2 intervals are created for the presented ROSACE task set. Hence, the costs of both functions remain the same. The interrupt service routine (ISR) overhead is architectural and hence should not change for

Function	Min	Max
update spare capacity (up_{sc})	2,655	10,145
update ready list (up_{ready})	3,500	9,115
next job selection (sel)	1,850	2,350
ISR overhead (ISR)	2,560	3,120

Table 7.3: Minimum and maximum runtime overhead per slot for single core execution in ns [61]

an implementation of slot-level randomization using the same simulator and operating system. Randomization is not part of slot shifting and as such not covered by the above measurements. As calculating random numbers for each slot is independent of parameters like the number of tasks or intervals, we assume a constant per slot overhead. Moreover, assuming an O(1) `get_length` implementation of the ready list, pruning random values to a list index remains a constant operation.

We calculate the maximum runtime overhead and, accordingly, the minimum runtime overhead as:

$$t_{ov,rand,max} = rand_{max} + up_{sc,max} + up_{ready,max} + sel_{max} + ISR_{max} \tag{7.1}$$

$$t_{ov,rand,min} = rand_{min} + up_{sc,min} + up_{ready,min} + sel_{min} + ISR_{min} \tag{7.2}$$

Using the measurements from Table 7.3 for equation 7.1 and assuming $rand_{max} = 5,000$ ns, the maximum runtime overhead results in $t_{ov,rand,max} = 29,730$ ns, which is around 3 percent of the assigned slot size of 1 ms in [61]. Keeping in mind that ROSACE uses 6 times faster cores than [61] and that execution time does not scale exactly linear with processor speed, we can approximate the runtime overhead for ROSACE. Therefore, we divide these values by 5 for a core with 1.2 GHz and approximate the maximum runtime overhead for ROSACE to be $t_{ov,rand,max} = 6,000$ ns.

Under the assumption that $rand_{min} = 2,000$ ns, the minimum runtime overhead results in $t_{ov,rand,min} = 12,565$ ns, which is around 1.3 percent of the slot size in [61]. Dividing these values by 5 as explained earlier, we approximate the minimum runtime overhead for ROSACE to be $t_{ov,rand,min} = 2,500$ ns.

7.2.2 Approximations for Offline Schedule-Diversification

For our second mitigation strategy, we again can make use of the overhead measurements done in [61], which we show in Table 7.4.

Function	Min	Max
next job selection (sel)	1,850	2,350
ISR overhead (ISR)	2,560	3,120

Table 7.4: Minimum and maximum runtime overhead for single core execution in ns [61]

At runtime, the scheduler performs a table lookup to select the next job after each slot. In contrast to the slot-level randomization scheduling algorithm, the overhead only consists of the next job selection and the interrupt service routine. At the end of the hyperperiod, we select the next offline constructed schedule randomly. For the randomization function, we use the same estimates for the execution time as shown in Section 7.2.1. We calculate best and worst case runtime overhead for selecting a precomputed schedule in MPARM as shown below.

$$t_{ov,prec,max} = rand_{max} + sel_{max} + ISR_{max} = 10470ns \tag{7.3}$$

$$t_{ov,prec,min} = rand_{min} + sel_{min} + ISR_{min} = 6410ns \tag{7.4}$$

Under the assumptions for the ROSACE case study as presented in Section 7.2.1, best and worst case approximated overhead results in 1300 ns and 2100 ns, respectively. Thus, around 1 percent of the chosen slot size is used for scheduling on both, ROSACE and the ARM simulator MPARM. The next section shows how and if the approximated overhead deviates from measurements, including the randomization, of a scheduler implementation on FreeRTOS.

7.3 Implementation and Runtime Overhead on FreeRTOS

We implemented both mitigation strategies, online slot-level randomization and offline schedule-diversification, as single core scheduling algorithms for FreeRTOS (kernel version 10.2.1) on a MicroZed 7020 Board. In addition, we implemented pure time-triggered scheduling as well to provide a baseline to determine the additional overhead introduced by the mitigation strategies. Our results are comparable to a multicore implementation, as we consider partitioned multicore scheduling for our mitigation strategies. Figure 7.2 shows a photo of the MicroZed board in question.

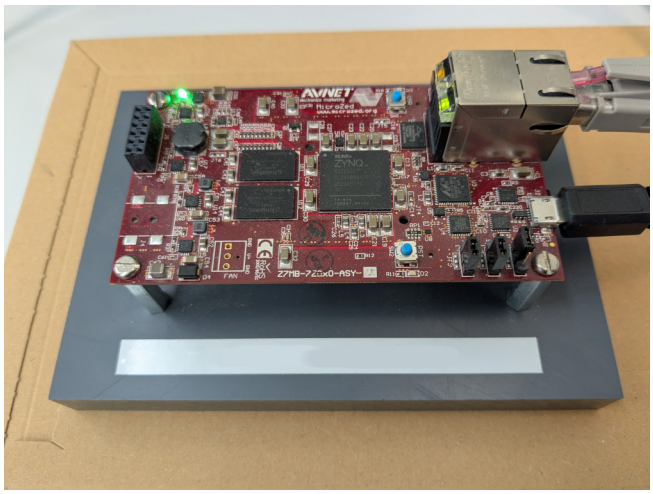


Figure 7.2: MicroZed 7020 Board

Since we implement an avionics case study, we chose hardware and software accordingly. FreeRTOS is an open-source real-time operating system for embedded devices [8] and used for avionics. The MicroZed board is a scaled-down version of the ZC706 ZYNQ-7000 evaluation board which is used in avionics research [29]. It features an ARM Cortex-A9 Core, a 32-bit processor implementing the ARMv7 architecture instruction set which runs at 667 MHz. Additionally, it provides a Performance Monitoring Unit (PMU) with a cycle count register which we use for our measurements.

We adapted the ROSACE case study for this platform and used its task set to measure the scheduling overhead of our mitigation strategies. In the following sections, we explain how we implemented online slot-level randomization, offline schedule-diversification, and time-triggered scheduling. We show how we conducted the experiments, i.e. runtime measurements, and present the measurement results. First, we start with the requirements of our scheduling algorithms.

7.3.1 Scheduling Algorithm Requirements

All three scheduling algorithms have different requirements before and during runtime. We denote the time before runtime as offline phase, and the time during runtime as online phase. During the **offline phase**, a single time-triggered schedule has to be constructed for time-triggered scheduling. This schedule resolves all dependencies and respects the timeliness requirements of its tasks. For offline schedule-diversification, we need not only one, but a set of feasible, distinct schedules. In the case of online slot-level randomization, we examine the task set parameters to determine the execution window of each task. Further, we use their deadlines to determine the capacity intervals and tasks belonging to each interval, and use their worst case execution times to calculate the spare capacity in each interval. Therefore, we need dedicated data structures which initialise and update capacity intervals and spare capacity. The tasks need to store the list of capacity intervals they belong to in their task control block, i.e. each instance of a task (job) belongs to a certain capacity interval.

During the **online phase**, all of the scheduling algorithms mentioned need a time base divided into time slots of fixed size. This requires a timer-based interrupt configured with the frequency needed to achieve the desired slot length. At the beginning of each slot, the scheduler must be invoked to decide on the next task to run based on its scheduling algorithm. Both in the pure time-triggered and the offline schedule-diversification approach, the scheduler chooses the next task according to the offline constructed schedule. For the former, this schedule repeats as long as the system is running. For the latter, a new schedule needs to be selected randomly for each hyperperiod (i.e. at the start of the hyperperiod). Online slot-level randomization takes the decision based on the spare capacity of the current capacity interval and a randomized choice of which ready task is executed next. For this matter, the data structures for capacity intervals and spare capacity need to be updated after every slot to reflect prior scheduling decisions and feasible task choices.

We summarize the requirements for each scheduling algorithm:

A. Pure time-triggered scheduling

- Time slots of fixed size using a timer-based interrupt
- An offline constructed time-triggered schedule
- Scheduler invoked at the beginning of every slot
- Task selection based on the time-triggered schedule

B. Offline schedule-diversification

- Time slots of fixed size using a timer-based interrupt
- A set of offline constructed time-triggered schedules
- Scheduler invoked at the beginning of every slot
- Task selection based on the time-triggered schedule
- Random selection of time-triggered schedule for each hyperperiod

C. Online slot-level randomization

- Time slots of fixed size using a timer-based interrupt
- Bookkeeping data structures for the capacity interval and spare capacities, including corresponding additions to the task control blocks
- Scheduler invoked at the beginning of every slot
- Task selection based on the spare capacity of the current capacity interval and a randomized choice of the ready tasks
- Update bookkeeping data after every slot according to scheduling decision

Some requirements of the three scheduling algorithms overlap. In order to summarize all requirements without duplicates, we present a numbered list which contains the requirements of all three scheduling algorithms:

1. Time slots of fixed size using a timer-based interrupt
2. Scheduler invoked at the beginning of every slot
3. Task selection based on current scheduling policy
4. Bookkeeping data structures
5. APIs for bookkeeping (to update bookkeeping data)
6. Offline constructed time-triggered schedule(s) and task set parameters

While the offline constructed schedules and task set parameters are formal requirements for the algorithm, for the following, we assume they already exist in the form of data. They do not need implementation. For the remaining requirements, requirements 1 to 5, we present their implementation in the following sections. First, we present already existing features of FreeRTOS and how we use them for our implementation. Then, we show our modifications of these features and additions to FreeRTOS in order to implement the three scheduling algorithms.

7.3.2 FreeRTOS Features

FreeRTOS provides many functions and mechanisms we can use to implement our scheduling algorithms and the case study. This section focuses on the functions and mechanisms already present in FreeRTOS that we use, whereas the next section details our additions and modifications. Pre-configured projects exist in the form of demo applications which already provide the user with correct FreeRTOS source file inclusions and the required references. Using this framework, a FreeRTOS project implementation can be split into five architectural layers [24], which we show in Figure 7.3:

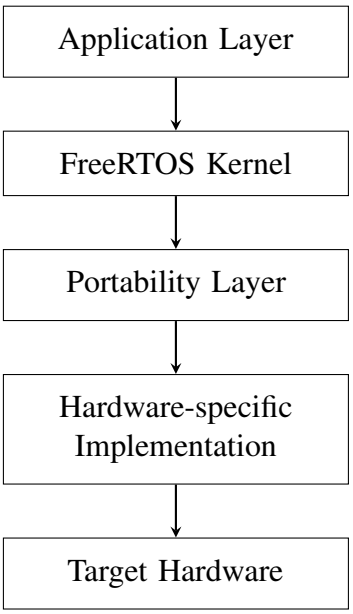


Figure 7.3: Architectural overview of a FreeRTOS project

The application layer contains implementations of tasks of the applications in question. The FreeRTOS kernel layer contains hardware-independent kernel features, such as task, queue, and list handling. The portability layer acts as interface between the kernel and the hardware platform. It provides standardized definitions for hardware- and application-specific implementations. The hardware-specific implementation layer consists of hardware-dependent kernel features, such as context switching, interrupt management, and timer management. These features are accessed by the FreeRTOS kernel through the standardized portability interface. Lastly, the target hardware layer represents the hard-

ware used with its vendor-specific functionality, which is the MicroZed 7020 board in our case.

Considering the requirements for our scheduling algorithms listed in Section 7.3.1, we present certain features of FreeRTOS which we used for their implementation [3, 59].

Requirement 1: Provide time slots of fixed size using a timer-based interrupt.
FreeRTOS provides a concept called tick where its real-time kernel measures the progression of time with strict accuracy using a tick count variable. Ticks divide continuous time into slices of the same length. The tick interrupt increments the tick count on each invocation which allows to slice time with a resolution of the chosen tick interrupt frequency. This mechanism allows to implement the time slots for our scheduling algorithms. In order to fulfill requirement 1, we need to use the following functions and variables FreeRTOS provides:

- *configTICK_RATE_HZ* is a compile time configuration constant that configures the frequency of the tick interrupt. For example, setting it to a value of 100 (Hz) results in time slices of 10 milliseconds.
- FreeRTOS also provides a macro called *pdMS_TO_TICKS* to automatically convert a value specified in milliseconds to the corresponding tick interrupt frequency. For the conversion, it uses the rate specified in *configTICK_RATE_HZ*. However, the macro is not able to create ticks for time slices smaller than 1 millisecond (i.e., a tick rate greater than 1000) [59]. Hence, we have to modify it according to our needs as we show in the next sections.
- The FreeRTOS kernel function *vTaskStartScheduler()* performs the steps required for scheduler as well as timer interrupt initialization. It includes a call to the hardware-specific implementation function *xPortStartScheduler()* which configures the duration and the function of the timer interrupt.

The tick interrupt handling is closely tied to scheduler initialization, as every tick interrupt needs to increment the tick count and to call the scheduler to take action. This leads us to our next requirement:

Requirement 2: Scheduler invoked at the beginning of every slot.
At every trigger of the timer interrupt, the hardware-specific interrupt service routine *FreeRTOS_IRQ_Handler()* calls the FreeRTOS kernel function *xTaskIncrementTick()*, which increments the tick count. It also controls the flag *ulPortYieldRequired* which indicates the need of a context switch. If the flag is raised, *FreeRTOS_IRQ_Handler()* saves the currently running task's context using the hardware-specific function *portSAVE_CONTEXT*, and then calls the FreeRTOS API *vTaskSwitchContext()* to decide on the next task for execution. FreeRTOS restores task context using *portRESTORE_CONTEXT*. Figure 7.4 illustrates how FreeRTOS handles a tick interrupt and which layers the interrupt handling involves.

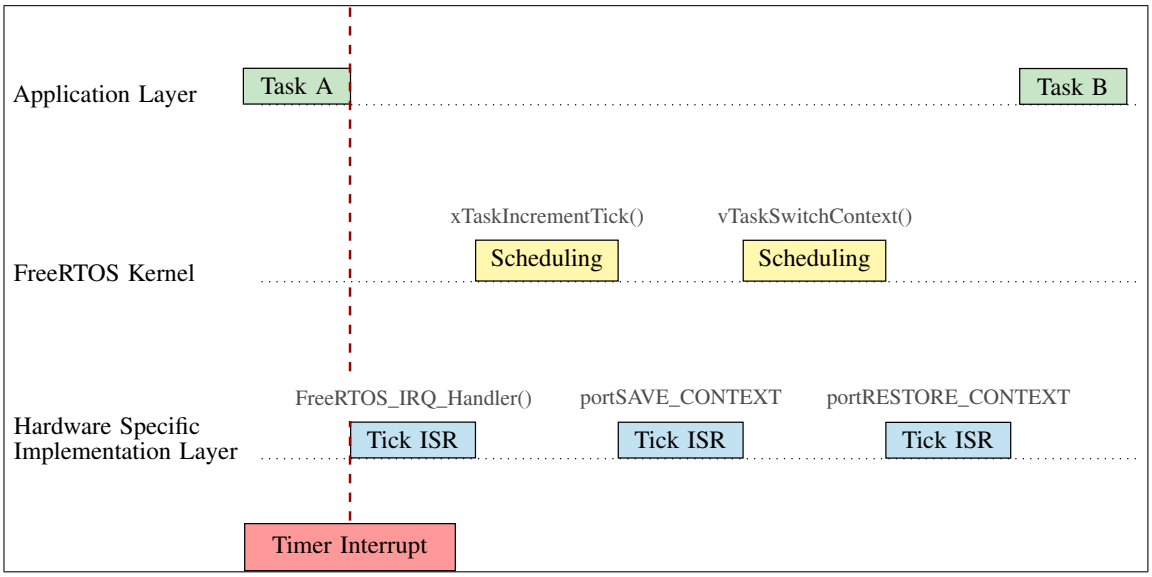


Figure 7.4: Handling of a timer interrupt in FreeRTOS

The timer interrupt marks the beginning of every slot. Since *xTaskIncrementTick()* is part of the kernel layer, we consider the scheduler invocation as part of its function. Task A, whose code resides in the application layer, executes until the periodic timer interrupt is triggered (or until the task terminates, whichever happens earlier). This results in a call to the tick interrupt service routine (ISR) which is part of the hardware-specific implementation layer. The following function calls necessitate switching between the hardware-specific implementation and kernel layer. After the tick ISR handler finishes execution and restores task context, the next task executes, task B in the case of this example.

Requirement 3: Task selection based on current scheduling policy.
The FreeRTOS kernel provides the function *vTaskSwitchContext()* as shown in Figure 7.4 to decide on the next task to execute. By default, it selects the next task for execution based on the API *taskSELECT_HIGHEST_PRIORITY_TASK* that chooses the next task based on its priority. Since our algorithms do not use priorities, we need to modify *vTaskSwitchContext()* based on the chosen scheduling algorithm. We present these modifications in the next sections.

Requirement 4: Bookkeeping data structures.
We need to create data structures for bookkeeping, including corresponding additions to the task control blocks. During initialization, the function *xTaskCreate()* creates a task control block (TCB) for each task. It requires certain parameters, such as the task name, code, and priority, and it adds created tasks to the ready task list. However, for bookkeeping purpose, we need to add additional fields to the TCB, e.g. the intervals a task belongs to. Since *xTaskCreate()* does not provide a method to add custom fields, we need to modify the TCB struct in the kernel. We present our modifications and the additional TCB fields in the next sections.

Additionally, we need a data structure for each capacity interval to keep track of the spare capacity and to be able to decide which task(s) we are allowed to execute next. This information resides within the scheduler. For each capacity interval in our hyperperiod, we thus create a data structure which consists of the following fields:

- Index, *index*: this static field is used as an identifier for the capacity interval.
- Start, *xStart*: this static field indicates the slot which marks the beginning of the capacity interval.
- End, *xEnd*: this static field indicates the slot which marks the end of the capacity interval.
- Ready task count, *xreadyTasks*: this count reflects the number of tasks that currently are in the ready state. It is updated whenever the state of a task changes to or from the ready state.
- Total task count, *xTotalTasks*: this static field indicates the total number of tasks that belong to the capacity interval.
- Spare capacity, *iSC*: this static field indicates spare capacity available for the capacity interval after offline calculations. We use this value to initialize the remaining spare capacity (the next field).
- Remaining spare capacity, *iSC_left*: this field reflects the currently remaining spare capacity. It is updated based on task execution, as explained in Section 5.1.3.
- Interval task list, *Int_tsklist*: this static field maintains a list of all tasks belonging to the capacity interval.
- Interval ready task list, *Int_readytsklist*: this field maintains a list of all tasks belonging to the capacity interval that currently are in the ready state. It is updated whenever the state of a task changes to or from the ready state.

We add these data structures to the scheduler in the initialization phase, during which we also create the FreeRTOS tasks. Static fields do not change after initialization, in contrast to non-static fields. For instance, while the index of an interval does and must not change, we need to update the field for the remaining spare capacity as it changes over time. This data structure is necessary for the bookkeeping function of online slot-level randomization and scales linearly with the number of different deadlines and thus capacity intervals in the system. As we show in a later section, its memory cost is small.

Requirement 5: APIs for bookkeeping.
We have to implement new APIs which are tied to task execution and the scheduler for bookkeeping purposes. Two functions already present in the FreeRTOS kernel provide useful APIs for this purpose:

- *xTaskIncrementTick()*, which provides mechanisms to check for newly arrived tasks,
- *vTaskDelayUntil()*, which we use to track when a task completes execution. Since *vTaskDelayUntil()* is called after (periodic) tasks finish their execution and puts these tasks into blocked state until the beginning of their next period, it allows to pinpoint the point in time when a task finishes execution.

Figure 7.5 summarizes and shows the FreeRTOS functionality we use and modify for our implementation. We continue to use this figure in the next sections to track the changes needed for implementing our scheduling algorithms on FreeRTOS.

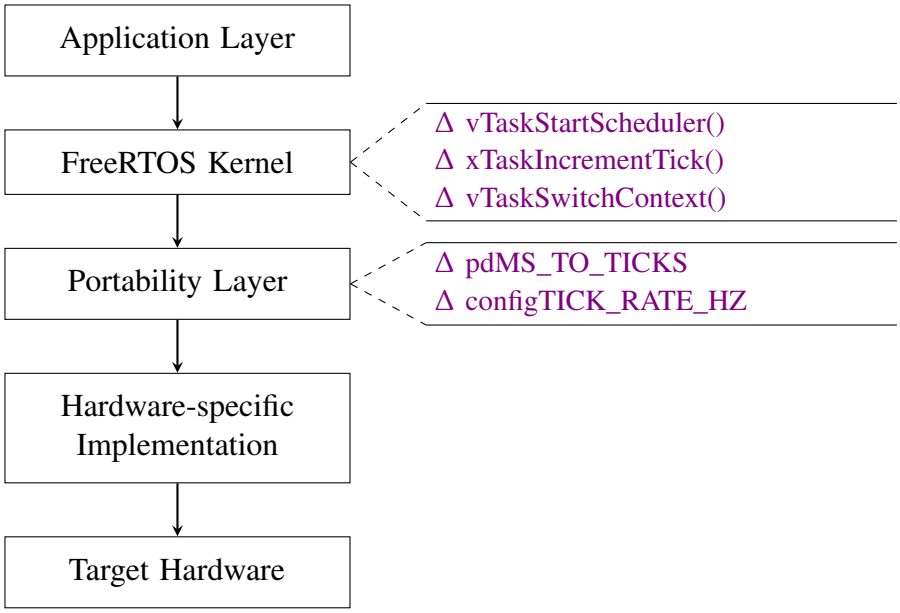


Figure 7.5: Changes to FreeRTOS functionality

We summarize our updated requirements for scheduler implementation which arise from existing as well as missing FreeRTOS features as follows:

1A. Time slices smaller than 1 millisecond.

3A. New task selection mechanisms (without priorities, according to the scheduling policy of our scheduling algorithm).

4A. Additional fields to the task control block for bookkeeping.

5A. New bookkeeping APIs (using *xTaskIncrementTick()* and *vTaskDelayUntil()*).

The next section details the implementation of these requirements using the FreeRTOS features we introduced earlier.

7.3.3 Scheduler Implementation

Building upon the FreeRTOS features introduced in the last section, we implement additional functionality in order to realize our three scheduling algorithms. First, we explain the implementation of new functionality according the updated requirements from the last section. Without modification, FreeRTOS does not or only partially meet our scheduling algorithm requirements. Then, we detail the implementation of the three scheduling algorithms and visualize them as flow charts.

Requirement 1A: Time slices smaller than 1 millisecond.
As discussed in Section 7.3.2, if the tick interrupt frequency (*configTICK_RATE_HZ*) is bigger than 1000 Hz, the macro *pdMS_TO_TICKS* cannot be used. The task set of the case study requires time slices smaller than 1 millisecond. Therefore, we redefine the macro *pdMS_TO_TICKS* in FreeRTOS. With our modification, it is able to accept time specified in microseconds as an argument and it converts the microsecond value into the corresponding tick interrupt frequency.

Requirement 3A: New task selection mechanisms.
When FreeRTOS initiates a context switch, it calls the context switch API *vTaskSwitchContext()* which resides in the kernel. *vTaskSwitchContext()* selects the next task for execution using the API *taskSELECT_HIGHEST_PRIORITY_TASK*, which selects the next task according to its priority. Since our algorithms do not use priorities, we exclude this API and implement our own task selection API, which we call *vSelect_Task()*. *vSelect_Task()* selects the next task based on the scheduling algorithm we use. For online slot-level randomisation, it randomly selects a task based on the spare capacity remaining for the current capacity interval and the ready tasks. For offline schedule-diversification, it randomly selects an offline-constructed schedule at the start of the hyperperiod and then continues to select the task according to the chosen schedule. For time-triggered scheduling, it selects the task according to the time-triggered schedule.

Requirement 4A: Additional fields to the task control block for bookkeeping.
For the online slot-level randomization scheduling algorithm, a data structure is necessary to store information about the capacity intervals and the tasks associated with them. We store this data structure as part of the scheduler in the FreeRTOS kernel layer as we described in Section 7.3.2. Additionally, we have to add fields to the task control blocks (TCBs) which also reside in the FreeRTOS kernel layer. The function *xTaskCreate()* creates the TCB for each task, but is not able to add custom fields. Therefore, we have to modify the TCB struct in the kernel that *xTaskCreate()* uses as template when creating tasks and their TCBs. In the following, we first present the TCB fields that are already present in the FreeRTOS kernel and then introduce and explain our additional fields for bookkeeping purposes.

FreeRTOS has the following standard fields in its task control block, which we describe on the basis of the FreeRTOS kernel code comments as they are not mentioned in the manuals:

- *pxTopOfStack:* A pointer to the location of the last item placed on the task's stack.
- *xStateListItem:* Denotes the state of the task (e.g. ready, blocked, suspended).
- *xEventListItem:* Used to reference a task from an event list.
- *uxPriority:* Priority of the task, where 0 is the lowest priority.
- *pxStack:* Points to the start of the stack.
- *pcTaskName:* Descriptive name given to the task during creation.

In addition to these standard fields, we add more fields to the task control block. Their addition is necessary to implement the bookkeeping mechanism of our online slot-level randomization scheduling algorithm. In order to determine more easily which task online slot-level randomization allows to execute next, we added the concept of an interval ready list which is also part of the capacity interval's bookkeeping data structure. The interval ready list comprises all ready tasks that belong to a certain (or the current) capacity interval, whereas the ready list, henceforth called "global ready list", means the ready list for all tasks. The following list introduces the additional TCB fields:

- *Tasktype:* Distinguishes between the IDLE task and other tasks; the distinction is necessary to correctly update the spare capacity after task execution.
- *uReadyIndex:* Holds the index of the task in the global ready list.
- *NumofInts:* Holds the number of capacity intervals to which the jobs of the task belong.
- *uMyIntervallist:* Holds a list of capacity intervals to which the jobs of the task belong.
- *uIntIndex:* Static array that holds the index of the task's jobs in each interval ready list. A static index removes the overhead of an array search and is beneficial when adding or removing a job from the interval ready list. Only when the job is ready, can it be found at its position as indicated by uIntIndex.
- *uJobInterval:* Indicates the capacity interval to which the current instance (job) of the task belongs to.

We add and initialize these new task control block fields during FreeRTOS task (and job) generation.

Requirement 5A: New bookkeeping APIs.
As discussed in the previous section, existing FreeRTOS features allow to check for newly arrived tasks and for tasks that have completed execution. We add new APIs to reflect these changes for our current capacity interval ready list, which we call *vUpdateReadyList()* for removing and *vUpdateTasklist()* for adding tasks. We also implement a new API, *vSpareCapacityUpdate()*, to update the spare capacities according to the task chosen for execution. Section 5.1.3 explains the calculation of the new spare capacity. These additions are necessary for the online slot-level randomization scheduling algorithm.

Now that we have fulfilled all requirements for our scheduling algorithms, we are able to implement them as part of the scheduler. We illustrate our implementation of each scheduling algorithm with a flow chart.

7.3.3.1 Time-Triggered Scheduler Implementation

In the case of time-triggered scheduling, we use a single offline constructed schedule. We do not need to calculate or initialize capacity intervals or spare capacity, and we do not use any bookkeeping mechanisms as we execute the chosen schedule without modification. Figure 7.6 shows the different actions the time-triggered scheduler performs. The scheduler is invoked with each tick increment interrupt and chooses the next task from the time-triggered schedule. After this scheduling decision, the execution of the scheduler ends with a context switch to the chosen task. No randomization occurs as time-triggered scheduling follows a strict and deterministic task execution order according to its schedule.

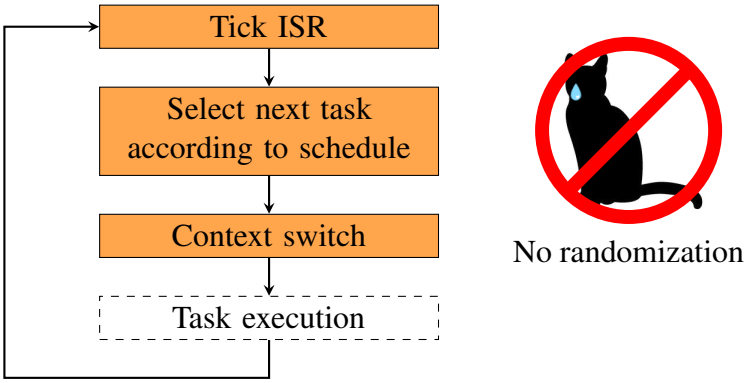


Figure 7.6: Time-triggered scheduler flow chart

7.3.3.2 Offline Schedule-Diversification Scheduler Implementation

In the case of offline schedule-diversification, instead of a single schedule, we have a set of offline constructed schedules available. We do not need to calculate or initialize capacity intervals or spare capacity, and we do not use any bookkeeping mechanisms as we execute the chosen schedule without modification. Figure 7.7 shows the different actions the offline schedule-diversification scheduler performs. Each tick increment interrupt invokes the scheduler. At the start of each hyperperiod, the scheduler chooses another schedule from the set of schedules at random. For the rest of the hyperperiod, it chooses the next task according to the chosen schedule. After this scheduling decision, the execution of the scheduler ends with a context switch to the chosen task.

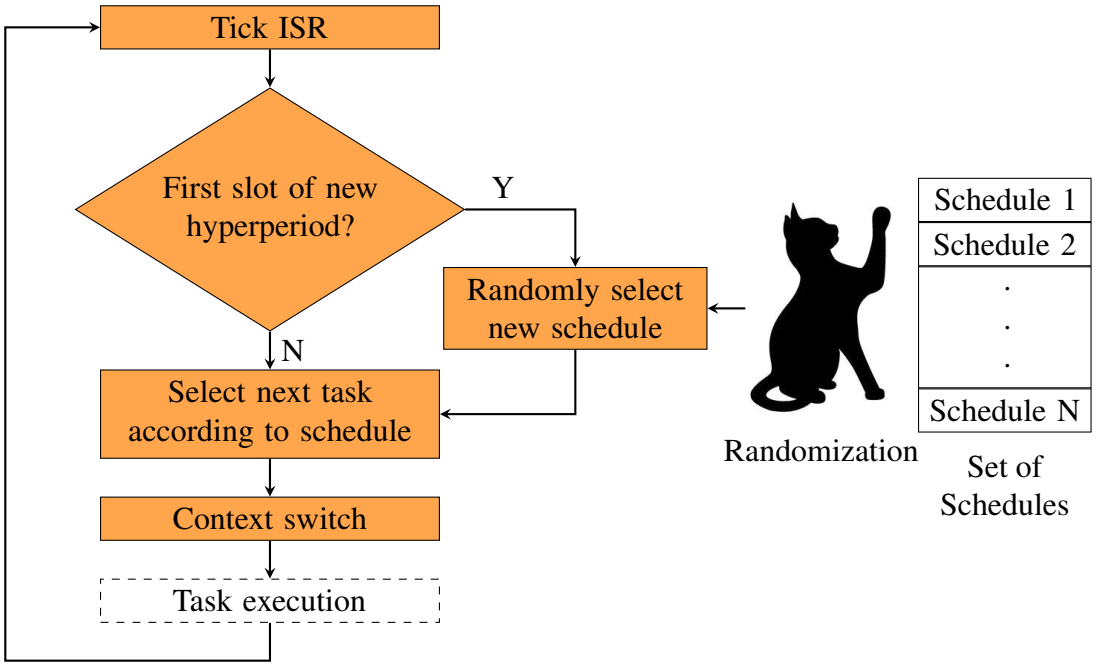


Figure 7.7: Offline schedule-diversification scheduler flow chart

7.3.3.3 Online Slot-Level Randomization Scheduler Implementation

For the online slot-level randomization scheduling algorithm, we first have to initialize our bookkeeping mechanisms. At system startup, our mitigation strategy therefore assigns capacity intervals and spare capacities based on the task set parameters, as we detail in Chapter 5.1. Upon task creation, we also initialize the fields of the task control blocks required for bookkeeping accordingly. After this initialization phase, each tick increment interrupt invokes the scheduler. The flow chart in Figure 7.8 shows the different actions

the online slot-level randomization scheduler performs upon each invocation after the initialization phase.

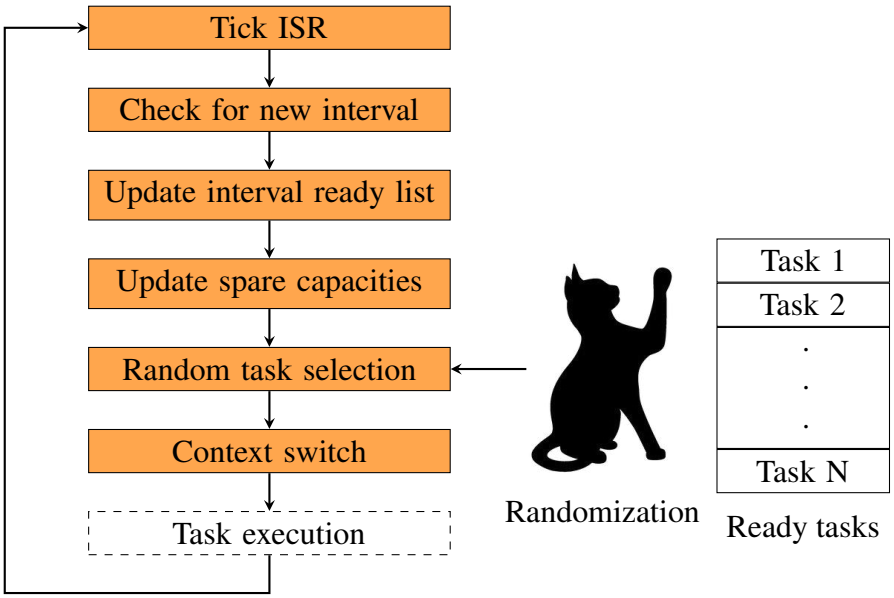


Figure 7.8: Online slot-level randomization scheduler flow chart

The scheduler performs this set of actions at the start of each slot and updates the capacity interval data structure accordingly throughout its execution. First, the scheduler checks if the capacity interval has ended in the previous slot by checking the static field *xEnd* of the capacity interval data structure. In this case, the scheduler resets the spare capacity of the previous capacity interval to its initial value to prepare it for the next hyperperiod, i.e it resets the value of *iSC_left* with the value of *iSC*. Resetting the spare capacities throughout the hyperperiod slightly decreases the worst case overhead as compared to a reset of all spare capacities at once at the hyperperiod end. The next action is the update of the interval ready list. If a task has finished in the slot before or if additional tasks have been released, the ready list is updated accordingly. This is done using the APIs *vUpdateReadyList()* and *vUpdateTasklist()*, respectively. Then, the scheduler updates the spare capacity *iSC_left* according to the scheduling decision taken in the previous slot using the API *vSpareCapacityUpdate()*. Chapter 5.1 details how to update the spare capacities after each scheduling decision. With the next and last step, the scheduler randomly chooses a task from either the global ready list or the interval ready list using the API *vSelect_Task()*. For the random selection, we use a pseudo-random number generator which adheres to the requirements we outline in Section 5.1.4. Whenever the spare capacity of the current capacity interval is less than one (i.e. zero or below as we allow only integer values), the scheduler randomly chooses only from the tasks belonging to the current interval. The execution of the scheduler ends with a context switch to the chosen task.

7.3.4 FreeRTOS Scheduler Implementation Summary

We implemented the major part of the scheduling algorithm functionality in the FreeRTOS kernel layer. Figure 7.9 summarizes our FreeRTOS scheduler implementation and also depicts the instances where we worked on other FreeRTOS layers. The color violet and the Δ symbol identify already existing FreeRTOS functionality that we modified for our purposes. Green and the + sign stand for APIs that we newly implemented. We indicate elements that FreeRTOS uses by default, but which we excluded for our application, in red and with a − sign.

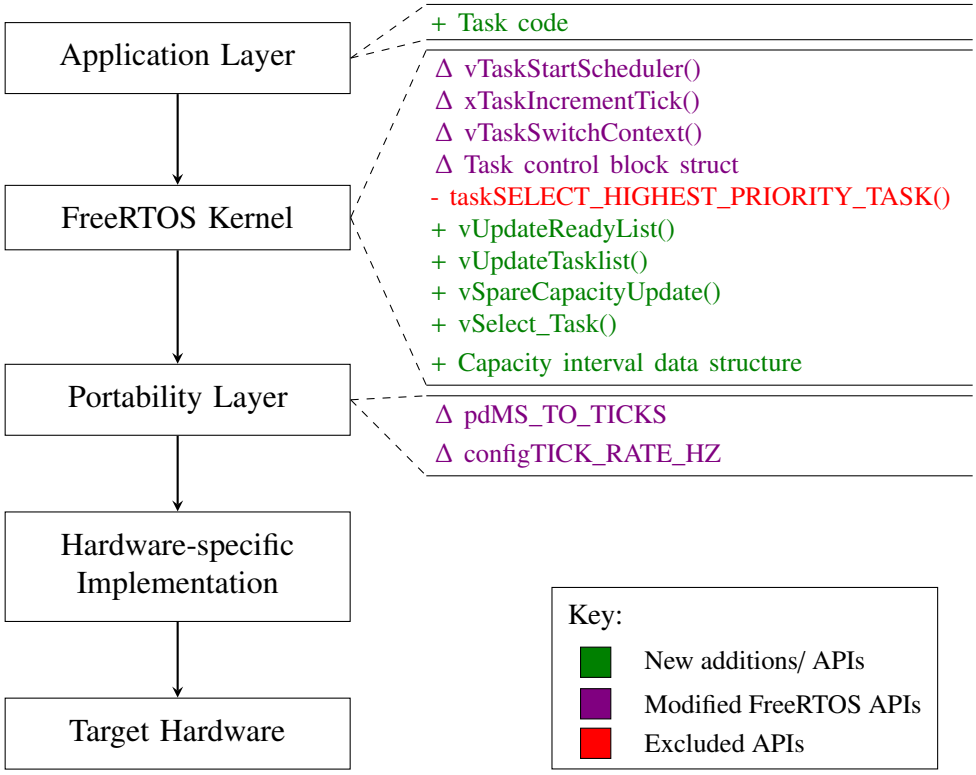


Figure 7.9: Architectural overview of changes to FreeRTOS functionality

We modified the kernel layer APIs *vTaskStartScheduler()*, *xTaskIncrementTick()*, and *vTaskSwitchContext()* to implement our scheduling algorithms. Since we do not use priorities, we excluded default task selection API, *taskSELECT_HIGHEST_PRIORITY_TASK()*, and replaced it with our own API, *vSelect_Task()*. We implemented additional APIs for the bookkeeping mechanisms of the online slot-level randomization algorithm in the FreeRTOS kernel, namely *vUpdateReadyList()*, *vUpdateTasklist()*, and *vSpareCapacityUpdate()*. Besides new bookkeeping APIs, we also added additional fields to the task's control block struct and a data structure for each capacity interval to enable the bookkeeping mechanisms. In the portability layer, we redefined the standard macro definitions of *configTICK_RATE_HZ* and *pdMS_TO_TICKS* for compatibility with our case study, i.e. to support smaller time slices than FreeRTOS allows. The code of our application tasks resides in the application layer. In the next section, we provide more details about the application tasks and explain how we adapted the case study and its task parameters.

7.3.5 Case Study Adaptation

We adapt the ROSACE case study introduced in Section 7.1 to our scheduler implementation while keeping periods and their implicit deadlines. Table 7.1 shows the task frequencies and worst case execution times of the original case study. The processor of the MicroZed board runs at 667 MHz. Following Schorr's suggestion of 200,000 clock cycles as slot shifting slot length [61], the slot size would be 300 μs. However, this value does not allow for an integer division of periods, thus we choose a slot size of 400 μs instead.

The authors of the ROSACE case study measured worst case execution times of their tasks on a CPU with double the frequency of our MicroZed board. As the task code is not available, we model their tasks using the parameters they provide. A rough estimate is a twice as high WCET on our board which would result in a WCET of 200 μs. Execution time, however, does not exactly scale linearly with the processor frequency. Measurements on ROSACE used a clock with the granularity of 100 μs. The tasks of this case study finished within that granule, which means that the WCET might be smaller than 100 μs.

Considering that ROSACE's WCETs were a result of 100 μs being the finest granule of their chosen clock and that we have slots double the size of our WCET estimate and four times their clock granule, we can safely conclude that ROSACE tasks finish execution within 400 μs on the MicroZed board. Thus, the WCET of the tasks equals one slot. However, this overapproximation of the slot size may also lead to more idle time of the CPU, i.e. unused processing time, especially if tasks finish earlier than their WCET. While this issue is common for real-time systems as tasks rarely execute for their WCET, it becomes more prominent when deciding on the slot size for a time-triggered system.

We model each ROSACE task as periodic FreeRTOS task with the above periods and worst case execution times in terms of slots. Since the task code is not available, we model the behaviour of the sensing and control tasks with arithmetic calculations. We ensure these calculations adhere to the same worst case execution time bounds as the ROSACE case study derived. When we apply the chosen slot size of 400 μs, Table 7.5 shows the resulting execution windows in terms of slots for each job within the hyperperiod.

7.3.6 Runtime Overhead Measurement Implementation

Both mitigation strategies and the time-triggered scheduling algorithm schedule the task set of the ROSACE case study that is presented in Table 7.5. The online slot-level randomization scheduler directly works with the task set parameters, whereas the offline schedule diversification scheduler works with a set of schedules derived from it. The time-triggered scheduler uses a single offline generated schedule. In order to measure the runtime overhead of our mitigation strategies and to compare it against the pure time-triggered scheduler, we measure the runtime of all three schedulers. We have to meet two requirements to be able to measure runtime:

1. Provide a runtime measurement mechanism
2. Provide markers for significant events

Name	Start	End	WCET
h_filter	0	25	1
az_filter	0	25	1
Vz_filter	0	25	1
q_filter	0	25	1
Va_filter	0	25	1
h_filter	25	50	1
az_filter	25	50	1
Vz_filter	25	50	1
q_filter	25	50	1
Va_filter	25	50	1
altitude_hold	0	50	1
Vz_control	0	50	1
Va_control	0	50	1

Table 7.5: Execution windows in terms of slots for FreeRTOS

Requirement 1: Runtime measurement mechanism.
As mentioned earlier, the ARM core on the MicroZed board features a performance monitoring unit (PMU) with a cycle count register. Reading the value from the PMU's cycle count register achieves cycle-accurate measurements [5]. We implement this function in Assembler language to minimize the measurement overhead and store the results in a C array. While its overhead is quite small, we also account for the additional overhead the measurement function introduces, as we will show later. Only after we finish running the experiments, we calculate the runtime overhead from the timestamps. This allows us to exclude these purely measurement-related calculations from the scheduling runtime overhead.

Requirement 2: Significant event markers.
FreeRTOS provides trace hook macros for data collection. They are empty macros at certain points of interest within the FreeRTOS source code which users can modify for application-specific tracing [3, 59]. On every trigger of the tick increment interrupt service routines, FreeRTOS invokes the trace macro *traceTASK_INCREMENT_TICK* which also marks the start of the slot. However, we do not need this event, as we explain in the next section. We did not find other trace hook macros for our remaining relevant scheduling runtime events. Thus, we added the measurement function at places in the code which correspond to the significant events.

Our runtime overhead measurement methodology is the same for all three scheduling algorithms as we measure the runtime of the scheduler. The next section presents our runtime overhead measurement methodology and implementation in more detail. Readers that are more interested in the results of the measurements may skip to Section 7.3.7.

7.3.6.1 Measurement Methodology

If we want precise measurements for runtime overhead, we need to understand how FreeRTOS operates and draw clear borders on which functionality belongs to scheduling and which does not. The tick interrupt service routine invokes the scheduler for the activities performed for the next slot. Slots, per definition, start once the tick is incremented, which happens a small amount of time after the tick interrupt service routine starts executing. As we illustrate in Section 7.3.2, the *FreeRTOS_IRQ_Handler()* API resides in the hardware-specific implementation layer and calls the FreeRTOS kernel API *xTaskIncrementTick()* which then increments the tick count. Thus, the measurement for the scheduling overhead of a single slot needs to include that small portion of the ISR before the start of the slot as per its definition. Figure 7.10 presents this overhead as scheduling overhead B and shows it in light blue in three instances.

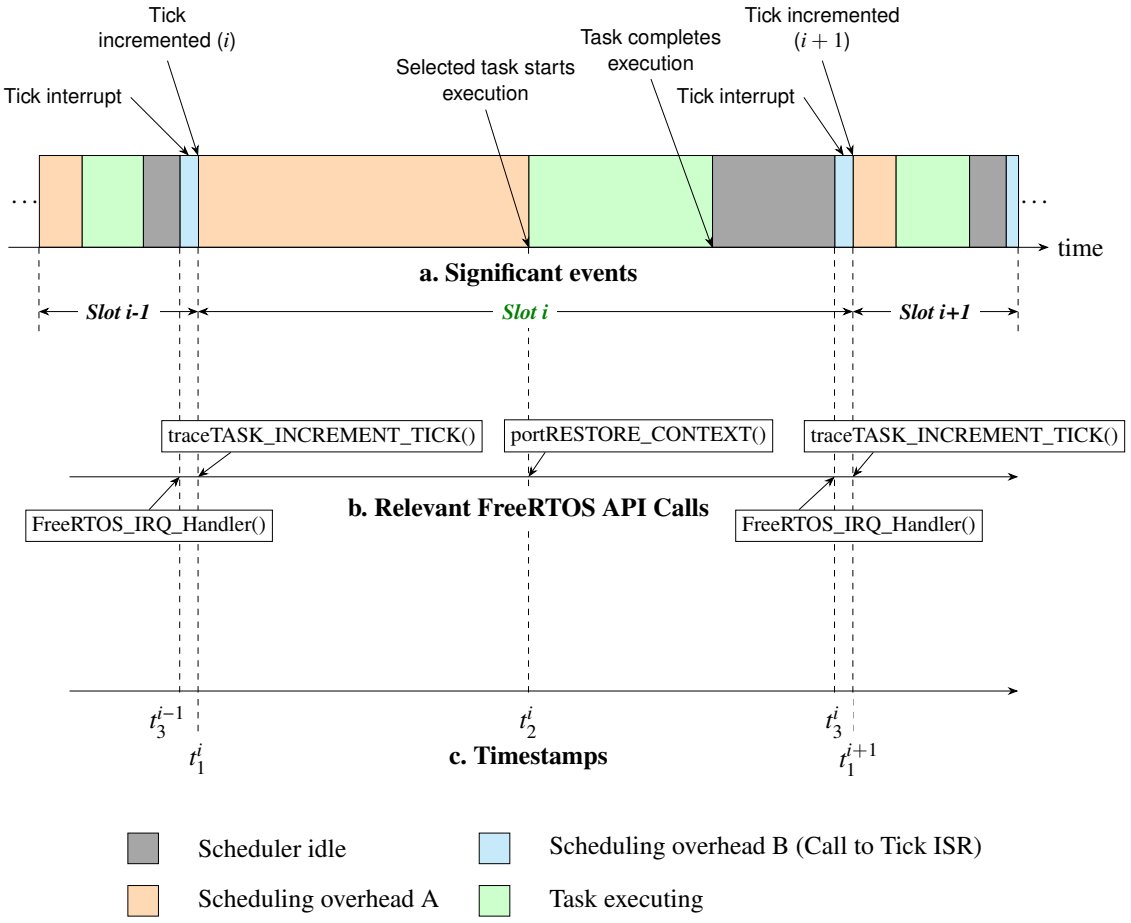


Figure 7.10: Scheduling runtime overhead measurement overview including significant events, timestamps and FreeRTOS API calls

Scheduling overhead B starts with the call to the API $FreeRTOS_IRQ_Handler()$ at times t_3^{i-1} and t_3^i, and ends at slot boundaries. New slots start when the tick count is incremented. In our example, these instances happen when the tick count value changes

from $i-1$ to i at time t_1^i and from i to $i+1$ at time t_1^{i+1}, respectively. Figure 7.10 further illustrates the scheduling runtime overhead measurement methodology in three layers:

a. Significant events, which includes events related to the ISR and scheduling events,

b. Relevant FreeRTOS API calls, which presents the APIs calls connected to the significant events, and

c. Timestamps, which we add to all significant events we need for our calculations.

We focus on slot i to explain our measurement methodology. The scheduling overhead that belongs to slot i consists of the scheduling overhead A after the tick increment from $i-1$ to i, and of scheduling overhead B before this tick increment. Figure 7.10 shows them as orange and blue regions. We focus solely on scheduling overhead and thus exclude task execution and idle times from our measurements, shown as green and grey regions, respectively.

Total Scheduling Overhead.
The total scheduling runtime overhead in each slot is the sum of scheduling overhead A and scheduling overhead B. For slot i, scheduling overhead A equals $t_2^i - t_1^i$ and scheduling overhead B equals $t_1^i - t_3^{i-1}$ as Figure 7.10 illustrates. We denote the runtime overhead of the scheduler in a given slot X during hyperperiod Y as $C_{sched}(X,Y)$. Equation 7.5 shows the calculation of $C_{sched}(X,Y)$:

$$C_{sched}(X,Y) = (t_2^X - t_1^X) + (t_1^X - t_3^{X-1}) = t_2^X - t_3^{X-1} \tag{7.5}$$

Thus, measuring t_1^X is not necessary. We only need two timestamps for overhead measurements, t_2^X and t_3^{X-1}, which reduces the probe effect. For a more detailed explanation on when these points in time occur, we use t_2^i and t_3^{i-1} from our example in Figure 7.10.

Timestamp t_2^i.
Scheduling overhead A starts with the tick increment. The scheduler executes until it hands over control to the task it chooses to execute next. For this purpose, the scheduler initiates a context switch by calling the FreeRTOS Kernel core API, *vTaskSwitchContext()*. Inside the context switch API, the scheduler calls the task selection API, *vSelect_Task()*, and selects a task according to its scheduling criteria. Execution returns to the interrupt service routine *FreeRTOS_IRQ_Handler()* and FreeRTOS loads the task context using the macro *portRESTORE_CONTEXT()*. We take timestamp t_2^i at the end of this macro, which marks the end of scheduling overhead A for slot i.

Timestamp t_3^{i-1}.
As Figure 7.10 shows, we need to include the $FreeRTOS_IRQ_Handler()$ API's runtime overhead for scheduling overhead B. We store the cycle count before we call this API, which corresponds to the timestamp t_3^{i-1}. More exactly, we take timestamp t_3^{i-1} before

executing the jump to the tick interrupt service routine from the interrupt vector table. Scheduling overhead B for slot *i* ends with the tick increment, which happens at timestamp t_1^i.

7.3.7 Runtime Overhead Measurement Results

We measure the scheduling runtime overhead in 100 runs of 100 hyperperiods (10000 hyperperiods in total) for each scheduling algorithm when scheduling the ROSACE task set, using FreeRTOS on the hardware platform presented in Section 7.3. Our measurements do not include initialization overhead of setting up the timers, creating capacity intervals, generating schedules or creating tasks as we consider these part of the offline phase. We extracted the best and worst case results from our measurements for each scheduling algorithm by taking its minimum and maximum cycle count of all runs. We also calculate the mean values by adding all measured values of all runs and then dividing that sum by the total number of values added together. Since cycles counts are integer numbers, we rounded the mean value up to next integer value. Table 7.6 presents the results for all three scheduling algorithms.

Scheduling Algorithm	**Runtime Overhead**								
	Cycle Count			**Microseconds** (μs)			**Percentage of Slot** (%)		
	Best	Mean	Worst	Best	Mean	Worst	Best	Mean	Worst
Online Slot-Level Randomization	1221	1538	5279	1.83	2.31	7.92	0.46	0.58	1.98
Offline Schedule-Diversification	1139	1232	3996	1.71	1.85	6.00	0.43	0.46	1.50
Time-Triggered Scheduling	1110	1178	3081	1.67	1.77	4.62	0.42	0.44	1.16

Table 7.6: Best, mean and worst case scheduling runtime overhead measurements of ROSACE task set scheduled with online slot-level randomization, offline schedule diversification, and time-triggered scheduling

We convert the cycle counts to their corresponding value in seconds for graphic visualisation. The processor runs at a frequency of 667 MHz, which means one clock cycle is about 1.5 nanoseconds long. Multiplied by the number of clock cycles, we achieve the second set of columns, which shows the scheduling overhead in terms of microseconds. Figure 7.11 visualizes the measurements as box plots.

The lower and upper whiskers present the minimum and maximum values of the measurements. The top of the box marks the 75th percentile whereas the bottom of the box marks the 25th percentile of all measurements. Thick horizontal lines stand for the median, i.e. the 50th percentile of each set of measurements, and might differ from the mean value. If no box is visible, the 25th and 75th percentile are too close to the median value for visual representation on a linear scale that includes maximum and minimum values.

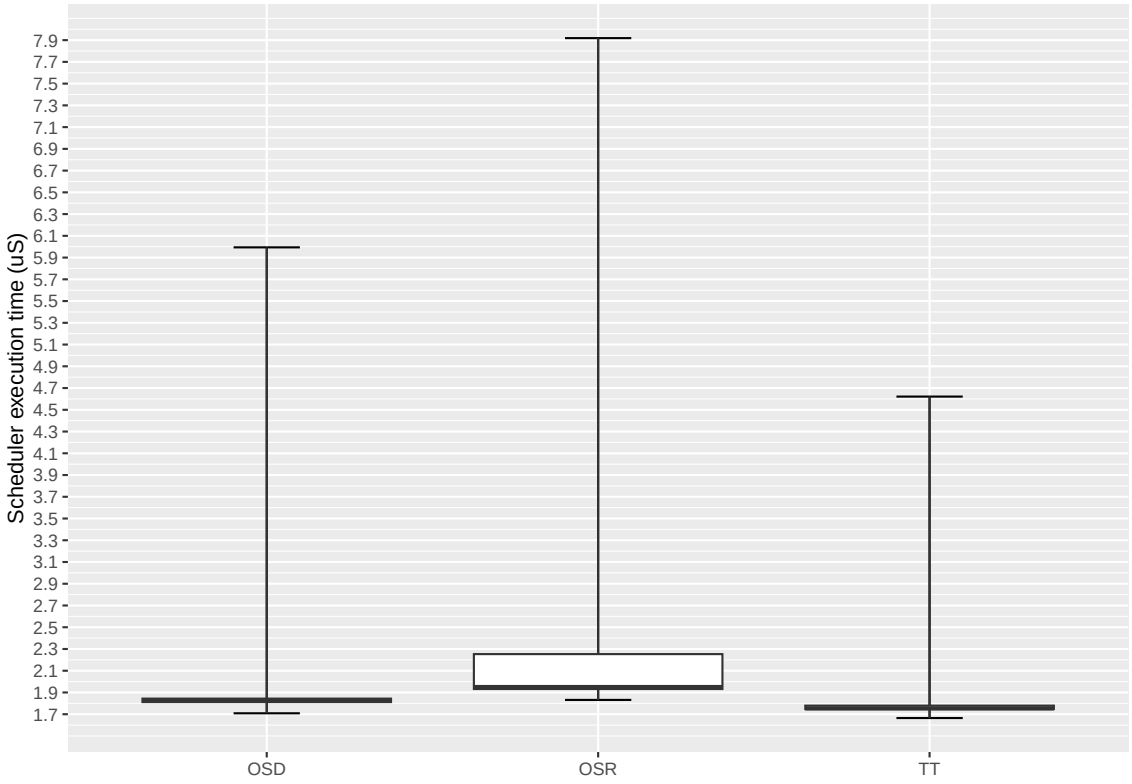


Figure 7.11: Visualization of scheduling runtime measurements (in μs) as box plots for online slot-level randomization (OSR), offline schedule diversification (OSD), and time-triggered scheduling (TT)

As both Table 7.6 and Figure 7.11 show, the best case runtime measurements for all three scheduling algorithms are similar. In the best case, the scheduling decision for all three algorithms is equal in complexity to a simple table-lookup without having to update spare capacities, ready tasks or switch schedules. The worst case runtimes differ more greatly. While the worst case of the offline schedule-diversification algorithm includes the selection of a new schedule for the next hyperperiod, online slot-level randomization not only has to randomly select a task but also update spare capacities, capacity intervals and the ready lists. Thus, while the offline schedule-diversification algorithm presents a 30 % increase over time-triggered scheduling in the worst case, online slot-level randomization has a 71 % increase over time-triggered scheduling. The mean and median values of the scheduling overhead as well as the 75th percentile remain close to the best case and similar to each other. While the worst case runtimes of online slot-level randomization and offline schedule diversification are greater than the worst case runtime of time-triggered scheduling, these three scheduling algorithms, on average, have comparable runtime.

Besides the runtime in absolute terms of cycle count or microseconds, it is also important to view the scheduling overhead in relation to the slot length. The scheduler must not take up much “space” of each slot, otherwise task scheduling becomes infeasible. Each

slot has a length of 400 μs, as we discuss in Section 7.3.5. The third set of columns of Table 7.6 presents the best, mean and worst case scheduling runtime overhead measurements in relation to this slot length, expressed as percentage. On average, the scheduling algorithms need about 0.5 % of the slot size, and remain within 2 % of the slot size for the worst case. These small numbers show that both mitigation strategies remain within reasonable bounds for scheduling tasks and remain comparable to time-triggered scheduling. As Figure 7.11 presents as well, even the worst case runtime for online slot-level randomization stays within 2 % of the slot size and is not even double the worst case runtime of time-triggered scheduling.

We also measured the runtime overhead of the new APIs, which we added to FreeRTOS for the online slot-level randomization algorithm, in 100 runs of 100 hyperperiods. The difference between two timestamps before the call to and after the end of each API returns the runtime overhead for our APIs. Table 7.7 shows the results for the APIs *vSpareCapacityUpdate()*, *vSelect_Task()*, *vUpdateTasklist()* and *vUpdateReadylist()*. For each API, we present the best, mean and worst case runtime overhead as cycle counts, in microseconds and in relation to the slot size (400 μs). Again, we rounded cycle counts to the next integer value, and mean and percentage values to the second decimal place.

API	**Scheduling Overhead**								
	Cycle Count			**Microseconds (μs)**			**Percentage of Slot (%)**		
	Best	Mean	Worst	Best	Mean	Worst	Best	Mean	Worst
vSpareCapacityUpdate()	201	229	282	0.30	0.34	0.42	0.08	0.09	0.11
vSelect_Task()	822	871	1098	1.23	1.31	1.65	0.31	0.33	0.41
vUpdateTasklist()	409	509	708	0.61	0.76	1.06	0.15	0.19	0.27
vUpdateReadylist()	819	901	1084	1.23	1.35	1.63	0.31	0.34	0.41

Table 7.7: Best, mean and worst case runtime overhead measurements of different parts of the online slot-level randomization algorithm

Figure 7.12 visualizes the runtime measurements from Table 7.7. The measurements show that the spread between best, mean and worst case runtimes is small, i.e. they are close together. Thus, our APIs have bounded runtime behaviour. The sum of the worst case runtimes of all APIs is notably smaller than the worst case runtime of online slot-level randomization (as shown in Table 7.7), because online slot-level randomization also calls native FreeRTOS APIs which add additional overhead. In the worst case, the four APIs have a combined runtime overhead of 3172 cycles, which is 60 % of the worst case runtime of online slot-level randomization. Since these APIs make up the core of its behaviour, this is an acceptable portion of its runtime overhead. On a similar note, the sum of the best case runtimes of all APIs is greater than the best case runtime of online slot-level randomization (as shown in Table 7.7). In the best case, online slot-level randomization does not need to call *vSpareCapacityUpdate()*, *vUpdateTasklist()*, or *vUpdateReadylist()* and thus has a lower runtime overhead.

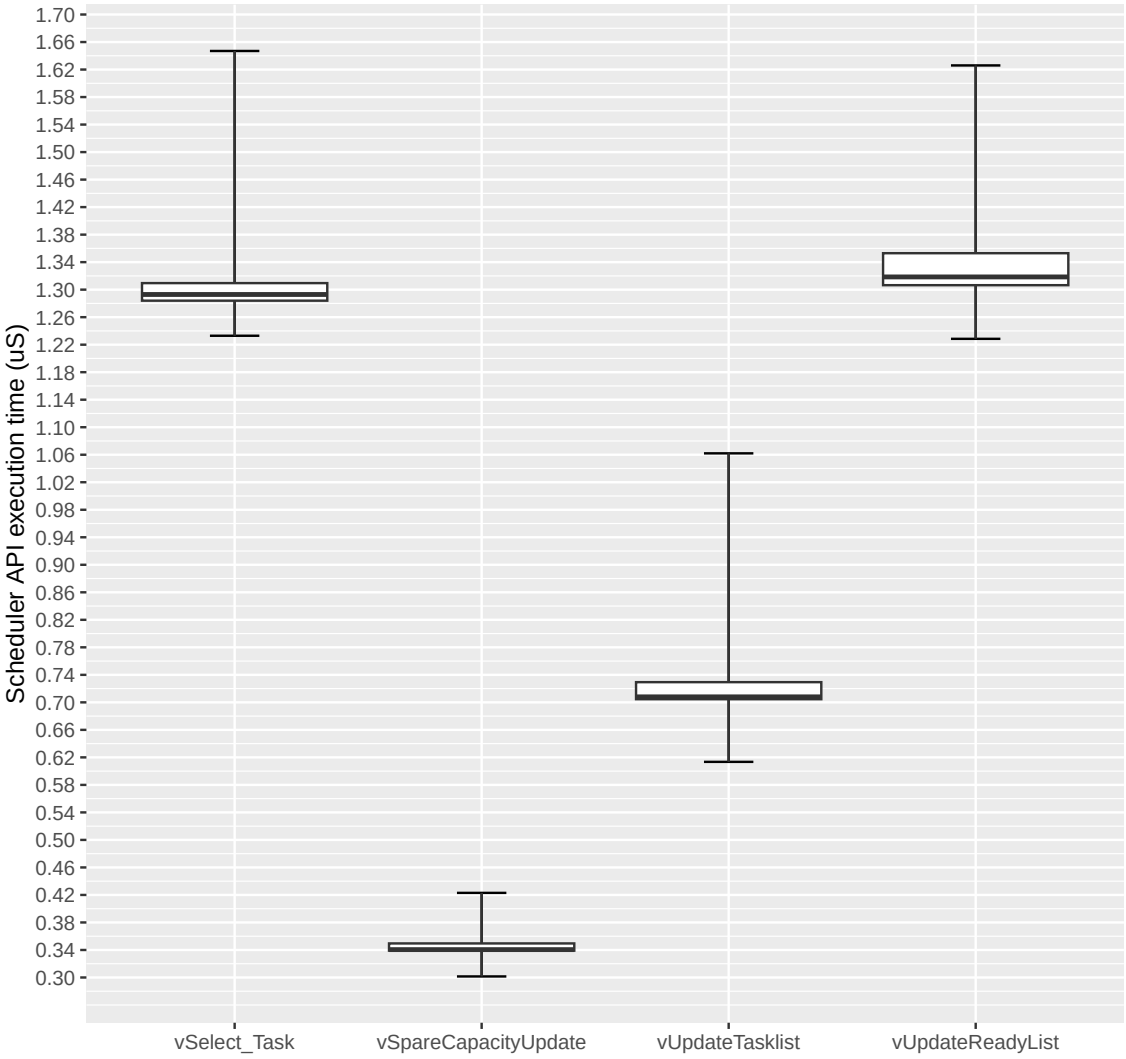


Figure 7.12: Visualization of API runtime measurements (in μs) as box plots

In the next section, we explore the probe effect of our measurements, i.e. their influence on the measured runtime overhead.

7.3.7.1 Accounting for the Probe Effect

Measurements typically add additional runtime overhead to the system and the functions they are analysing. We therefore took great care to incur as little additional runtime overhead for measurements as possible. In order to minimize the probing effect, we implement the cycle counts measurement as Assembly instructions. We use the MRC instruction to obtain the value of the cycle count register of the performance measurement unit. This instruction causes a coprocessor to transfer a value to an ARM core register [4]. We then store the obtained cycle count inside a C-array where we store all cycle counts of the cur-

rent experiment. Only after the experiment finishes, i.e. after we schedule the task set for a set number of hyperperiods, do we calculate the corresponding times for the scheduler execution.

ARM does not provide a cycle timing for the MRC instruction [5]. Therefore, we measure the timing of the MRC instruction over 5000 runs of the ROSACE task set and obtain a maximum, minimum and mean value of 4 clock cycles. We proceed in the same way for writing to the C-array and obtain maximum, minimum and mean values of 26, 18 and 19 clock cycles, respectively. Additionally, we need to save registers on the stack before measuring and restore them afterwards to prevent overwriting application data. These instructions are load and store operations, for which the instruction cycle timings are 2 cycles each [4]. Table 7.8 shows the total runtime overhead of the measurement function in the best, mean and worst case.

	Best	**Mean**	**Worst**
Save context	2	2	2
MRC instruction	4	4	4
C-array write	18	19	26
Restore context	2	2	2
Total:	26	27	34

Table 7.8: Best, mean and worst case runtime overhead of measurement function parts, measured in clock cycles

The order of the instructions in Table 7.8 reflects the implementation of the timing measurement function. Since we take two timestamps during each execution of the scheduler, we add one full cycle of the measurement function to the scheduling runtime overhead. For the first timestamp, t_3^{i-1} in the example of Section 7.3.6.1, the instructions following taking the timestamp add to the scheduling runtime overhead, namely the C-array write and restoring the context. In case of the second timestamp, t_2^i according to Section 7.3.6.1, the instructions until taking the timestamp add to the scheduling runtime overhead, namely saving the context and the MRC instruction. Thus, the probe effect of the measurement itself adds 26 clock cycles in the best case, 34 clock cycles in the worst case, and 27 clock cycles on average.

We note that measuring the probe effect also incurs a probe effect, i.e. these measurements include the probe effect itself. Therefore, we conclude that the total values of Table 7.8 provide a safe upper bound. Since these values vary and their best, mean and worst case may be different from the best, mean and worst case of the overall scheduling runtime overhead, we do not deduct them from the results presented in Table 7.6 and Table 7.7. However, the overall probe effect in terms of runtime overhead is low.

7.4 Memory Requirement

This section discusses the additional memory overhead introduced by our mitigation strategies compared to time-triggered scheduling with a static schedule. Both offline schedule-diversification and online slot-level randomization need additional data structures. Offline schedule-diversification needs to store more than one schedule, whereas online slot-level randomization needs bookkeeping data structures. Our multicore mitigation strategies have the same memory requirements per core as their single core counterparts since we consider a partitioned multicore scheduling strategy. In the following, we show that the memory requirement for each of the mitigation strategies is low.

7.4.1 Offline Schedule-diversification

Each precomputed schedule needs to be stored in memory. Offline constructed schedules for time-triggered systems only need two pieces of information: which instance of a task is executed in which slot. Table 7.9 shows an example of a time-triggered schedule where IDs are unique identifiers for task instances and ID 0 refers to the idle task.

Slot	0	1	2	3	4	5	6	7	8	9	10	11	...
ID	1	0	3	2	0	0	0	4	0	0	2	0	...

Table 7.9: Exemplary time-triggered schedule

An array whose index stands for the slot in question and whose values correspond to the IDs is able to represent a schedule. For our experiments, we use a two-dimensional array to store the set of offline constructed schedules. We refer to it as $schedule set[M][N]$ in order to calculate the memory cost.

The length of the hyperperiod defines the size of its first dimension. For our adaptation of the ROSACE case study, the hyperperiod consists of 50 slots. Thus, the dimension M has the size 50. Each element in the array is an unsigned 8-bit integer and thus needs 1 byte of memory. A single schedule for ROSACE thus has a memory requirement of 50 bytes.

The total amount of memory needed depends on the number of schedules stored for offline schedule-diversification, as the number of schedules defines the size of the second dimension, N. Thus, if we want to store 1000 schedules for the ROSACE task set, $schedule set[50][1000]$ needs 50kB of memory. In comparison, time-triggered scheduling only uses one schedule, i.e. 50 bytes of memory.

7.4.2 Online Slot-Level Randomization

Online slot-level randomization needs bookkeeping data structures. They allocate additional space in memory, especially compared to time-triggered scheduling with a single schedule. We implemented these bookkeeping data structures as extension to the task control blocks and in the form of data structures which represent the capacity intervals. In this

section, we analyse both the task control block extension and the capacity interval data structure, and calculate their allocated space in memory. We use ARM's architecture reference manual [4] to determine the size of different data types in the processor. In addition, we confirmed these values with the results of the sizeof() operator. Table 7.10 shows the additional TCB fields and lists their memory requirements:

Data type	**Field**	**Size**
uint8_t	Tasktype	1 byte
uint8_t	uReadyIndex	1 byte
uint8_t	NumofInts	1 byte
uint8_t	uJobInterval	1 byte
uint8_t*	uMyIntervallist[M]	$4 + M \times 1$ bytes
uint8_t*	uIntIndex[M]	$4 + M \times 1$ bytes
	Total:	$12 + M \times 2$ bytes

Table 7.10: Memory requirement of the task control block extension, where M is the number of capacity intervals

The uint8_t* data type is a pointer which, according to the architecture reference manual [4], is 4 bytes in size. Both pointers in the TCB extension point to uint8_t arrays of size M which are part of the TCB extension. Every task has their own task control block, so the total of Table 7.10 needs to be multiplied by the number of tasks in the system. For the ROSACE task set in our experiment, we have 8 tasks in the system and need two capacity intervals. Thus, the total memory requirement for the TCB extensions amounts to $(12 + 2 \times 2) \times 8 = 128$ bytes for the ROSACE task set.

Table 7.11 lists the memory requirement of capacity interval data structures. The size of each capacity interval in memory scales linearly with the number of tasks (N) belonging to it. While the ARM processor in use supports the storage of four 8-bit (1 byte) integers in one of its 32-bit registers, it needs to pad single 8- or 16-bit integers with zeros or sign-extension [4]. Therefore, we need to also consider 3 bytes of padding for the interval data structure. TCB_t* are arrays of size N whose elements are pointers to the task control blocks of the tasks they refer to, thus they require multiples of 4 bytes in memory.

As mentioned earlier, the ROSACE task set requires two capacity intervals for bookkeeping. Five tasks belong to the first and eight tasks to the second capacity interval. Thus, the total memory requirement for the capacity interval data structures amounts to $16 + 8 \times 5 + 16 + 8 \times 8 = 136$ bytes for the ROSACE task set.

In the worst case, every job in the system has its own execution window and capacity interval. Then M, the number of capacity intervals, is equal to the number of jobs, and N, the number of tasks belonging to a capacity interval, is 1 for every capacity interval.

Data type	Field	Size
Structure padding		3 bytes
uint8_t	index	1 byte
uint8_t	xStart	1 byte
uint8_t	xEnd	1 byte
uint8_t	xreadyTasks	1 byte
uint8_t	xTotalTasks	1 byte
int32_t	iSC	4 bytes
int32_t	iSC_left	4 bytes
TCB_t*[N]	Int_tsklist	$4 \times$ N bytes
TCB_t*[N]	Int_readytsklist	$4 \times$ N bytes
	Total:	$16 + 8 \times$ N bytes

Table 7.11: Memory requirement of capacity interval data structure, where N is the number of tasks belonging to the capacity interval

If we denote the number of tasks with n and the number of jobs with j, the worst case assumption of the additional memory requirement equals:

$$memcost_{OSR,WC} = (12 + 2j) \times n + 24 \times j[bytes] \tag{7.6}$$

However, with knowledge about task set parameters, and the resulting execution windows and capacity intervals, the memory overhead may be significantly lower than this worst case assumption. For ROSACE, with 8 tasks and 13 jobs in total, this worst case assumption amounts to 616 bytes. The actual memory requirement that we calculated earlier is $128 + 136 = 264$ bytes, which is only around 43 % of its worst case assumption, proving it to be pessimistic. We are able to conclude that online slot-level randomization has a low memory overhead, which is around 4-5 times greater than the memory overhead of time-triggered scheduling in our experiments.

7.5 Solution Space (number of valid schedules)

In order to show how many possible schedules slot-level randomization covers, we calculate the total number of feasible schedules. First, we calculate the total number for the task set presented in Table 7.2 used for the MPARM simulator. For each execution window, the binomial coefficient $\binom{n}{k}$ calculates the number of possibilities to execute the task in different slots, where n is the window size and k the worst case execution time, both quantified in slots. The binomial coefficients of neighbouring and overlapping execution windows are multiplied with each other. If execution windows overlap, we subtract the worst case execution time of tasks belonging to execution windows whose binomial coefficients are

already accounted for in the equation ("preceding" binomial coefficients) from the window size. Thus, we calculate the number of possible feasible schedules for the presented task set as shown below. On the left side of the equation, the binomial coefficients of the five tasks with periods of 50 slots are calculated two times, because the hyperperiod results in 100 slots. Their combined worst case execution time of 10 slots is then subtracted from the execution window sizes of the tasks with a period of 100 slots.

$$\left[\binom{50}{1}\binom{49}{1}\binom{48}{1}\binom{47}{1}\binom{46}{1}\right]^2 \times \binom{90}{1}\binom{89}{1}\binom{88}{1} = 4.56 \times 10^{22} \tag{7.7}$$

Secondly, we calculate the total number of feasible schedules for the ROSACE case study on FreeRTOS, using the task set presented in Table 7.5. Due to the lower frequency of the processor running the implementation of our scheduler on FreeRTOS, we only have half the slots available compared to MPARM. The total number of feasible schedules in this case amounts to:

$$\left[\binom{25}{1}\binom{24}{1}\binom{23}{1}\binom{22}{1}\binom{21}{1}\right]^2 \times \binom{40}{1}\binom{39}{1}\binom{38}{1} = 2.41 \times 10^{18} \tag{7.8}$$

4.56×10^{22} schedules with 50 bytes require 2^{81} bytes of storage and 2.41×10^{18} schedules of the same size 2^{67} bytes. Thus, we can safely conclude that it is infeasible to track or store all possible schedules in terms of memory space and computation time needed. Positive spare capacities, i.e. leeway in the schedule, are key for a high number of distinct feasible schedules.

7.6 Schedule Diversity

We introduced schedule entropy in Chapter 5.2.2 as a metric to quantify the diversity between schedules. This metric uses an approximation of the Shannon entropy and can be used to quantify the diversity of a set of schedules. In [43], we investigate the fundamental limitations to the achievable diversity imposed by the task set characteristics. There, we show how to determine the maximum upper-approximated entropy for the ROSACE task set with a slot size of 200 μs, as shown in Table 7.2. With this information, we calculate the minimal set of schedules to achieve this maximum upper-approximated entropy. In case of ROSACE, only a minimal set of 100 schedules is needed.

While this small number sounds tempting, especially with respect to the memory requirement of offline schedule-diversification, schedule entropy or the schedule diversity does not quantify security per se. If a set of schedules is more diverse than another set of schedules, the schedules have of the former have less similarities than the latter. Thus, the more diverse the schedules are, the lower is the probability that a job executes in the same slot over several schedules. This gives an attacker less information about the job execution itself, according to the Shannon theorem. However, using this metric also ensures that the schedules are easier to distinguish from one another. Depending on the memory and side

channel bandwidth available to the attacker, a set of only 100 diverse schedules might be easy to figure out and store. In Section 7.4.1, we show that a single schedule needs only 50 bytes of memory which substantiates this claim. Then, especially if the schedules are maximally diverse from each other, the schedule chosen online might be easy to recognize during execution and the attacker might execute a directed attack nonetheless.

Entropy rightfully has its place as security metric in literature, however, in our case, two important properties need careful consideration: periodicity and the attacker's available memory. Online slot-level randomization is able to explore the full solution space of feasible schedules while offline schedule-diversification is restricted to its set of schedules. Schedules repeat after their hyperperiod and are limited in number. If an attacker has enough memory available, which is likely due to the small schedule size, they might be able to store the set of diverse schedules. Diverse schedules become easy to recognize if the set of schedules is small since they are more different from each other than a less diverse set of schedules. However, if the set of schedules is large, enough similarity between the schedules remains to impede predictions.

We conclude that schedule diversity is an important metric to check for equal distribution of jobs over the slots in which they execute and thus impedes predictions about the slot they execute in, considering multiple hyperperiods. However, a large set of schedules is of even greater importance as it prevents predictions even when the attacker has enough memory available. Each additional schedule also means more time that the attacker has to spend eavesdropping through a side-channel which in turn means more time available to detect the intrusion.

7.7 Determinism Granularity

Time-triggered systems are characterized by their determinism with regard to task execution. Determinism, however, has different levels of granularity at which something may be called deterministic. Absolute determinism with respect to task execution is not possible due to physical constraints [28]. For example, clocks run at different speeds due to inherent physical differences and slight variations remain when manufacturing the processor. Even at task level it is generally not possible to predict which instructions of a task will be executed during runtime. They vary depending on the input and processor state, e.g. depending on which branches are taken. For this reason, task execution times vary during runtime. Therefore, the determinism granularity considered for time-triggered systems is at slot level. For each slot, we are able to determine the assigned task, and if we adhere to the (feasible) time-triggered schedule which assigns tasks to slots, we are able to meet all temporal constraints.

Our mitigation strategies do not allow for slot level determinism. Their very intention is to prevent predictions about which task will execute in which slot by using randomization. However, they allow for determinism at execution window level. Both mitigation strategies guarantee that each task finishes execution within its execution window. Thus our approach is deterministic at execution window level. Figure 7.13 illustrates the difference between slot level and execution window level determinism.

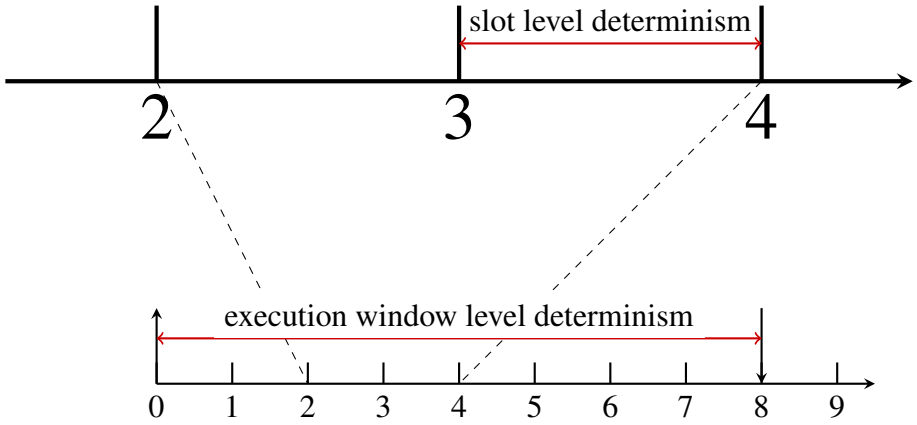


Figure 7.13: Different levels of determinism: top shows slot level, bottom execution window level

7.8 Discussion

Real-time systems are often implemented as embedded systems. As such, they are not only subject to size, weight and power considerations, but also have only limited memory available. Low memory cost and low computation overhead become even more important for these constrained systems. We analyse our mitigation methods with respect to these constraints and compare them to time-triggered scheduling in terms of overhead.

We used ROSACE for our evaluation which is a case study of a longitudinal flight controller of an aircraft. After demonstrating how to convert its task set parameters into execution windows and worst case execution times in terms of slots, we approximate the runtime overhead of our mitigation strategies on the MPARM simulator. For offline schedule-diversification, the minimum and maximum runtime scheduling overhead amounts to around 0.7 and 1.1 percent of the chosen slot size, respectively. For online slot-level randomization, using the same slot size, the minimum and maximum runtime scheduling overhead amounts to around 1.3 and 3 percent, respectively. These approximations based on measurements of previous work show promising, low runtime overhead which we were able to confirm with an implementation on FreeRTOS.

We implemented not only our two mitigation strategies, online slot-level randomization and offline schedule-diversification, but also time-triggered scheduling on FreeRTOS. Both mitigation strategies prove to be practical, as their measured runtime overheads in Section 7.3.7 show. Online slot-level randomization uses less than 2 percent of the slot for scheduling in the worst case. This result is lower than our approximated worst case runtime overhead and not even twice the worst case of time-triggered scheduling. The worst case runtime overhead for offline schedule-diversification is even lower which we expected since slot-level randomization has to take more complex decisions online. Using offline constructed schedules reduces the runtime overhead to 1.5 percent of the slot size. Time-triggered scheduling needs 1.2 percent of the slot size in the worst case, which means the runtime overhead of our mitigation strategies is comparable to time-triggered scheduling with a static schedule. We also account for the probe effect of our measure-

ments, which is quite low in terms of runtime overhead as it remains under 1 percent of the measured value on average.

Physical memory capacity limits the number of offline constructed schedules that can be stored in a system. However, the schedules themselves prove to be small with 50 bytes of memory needed to store a single schedule of a real-world task set comprising 13 jobs. For our case study, 1 MB would already be sufficient to store nearly 21000 different schedules. Schedule size and memory overhead in general increase with the number of jobs involved. If memory capacity is very limited, it is also possible to offload schedules to secondary storage by accepting an increase of scheduling overhead when loading the selected schedule at the beginning of the hyperperiod.

As we mention in Chapter 3, an attacker might identify a small number of schedules after several minutes or a few hours even for side channels with low bandwidth. Even under the assumption that the attacker is able to store a huge number of schedules, the higher the number of precomputed schedules, the longer it takes for the attacker to be sure which schedule currently is in use. Using schedule diversity as a metric to choose a smaller, more diverse set of schedules, however, might benefit the attacker as it might be easier to recognize single schedules. We rather encourage the use of a larger set of different schedules. Updating the stored set of schedules, especially if secondary storage is available to the system, is another method to mitigate the threat that the attacker might eventually identify the schedule in time. The threat is fully mitigated with slot-level randomization, which we recommend in general, due to the comparable overhead, and for systems with strict memory constraints.

Depending on the number of jobs, online slot-level randomization also has an inherent memory overhead due to its bookkeeping data structures. In case of our implementation of the ROSACE case study on FreeRTOS, this additional memory overhead amounts to 264 bytes, which is roughly five times more than a single schedule. Online slot-level randomization is, however, able to explore all feasible scheduling decisions and eventually covers the full solution space. For our case study, depending on the slot size, we show that there exist between 10^{18} and 10^{22} different feasible schedules. Storing all feasible schedules with today's memory capacities is impossible, especially when considering embedded systems. The ROSACE case study features a lot of leeway in their schedule. In other scenarios with higher task utilization, less room for randomization may exist. As mentioned earlier, positive spare capacities, i.e. leeway in the schedule, are key for a high number of distinct feasible schedules.

Randomization of tasks does not allow for the same level of determinism time-triggered scheduling offers. In time-triggered scheduling, for each slot, the task executing in that slot is known before runtime. However, we identified this slot level determinism as vulnerability of time-triggered systems. Online slot-level randomization impedes predictions about the slot in which a task executes through randomization on a slot-by-slot basis. While this mitigation strategy does not allow for slot level determinism, it allows for determinism at execution window level. In the case of offline schedule-diversification, while we do not know which schedules it chooses for each hyperperiod before runtime, during each hyperperiod, we know which task will execute in which slot. Therefore, offline schedule-

diversification offers more predictability than online slot-level randomization if we consider single hyperperiods.

8

CONCLUSION

Real-time systems need to not only provide correct output, but they must also provide output within certain timing constraints. If timing constraints are violated, the consequences range from a degradation of the quality of service provided up to catastrophic consequences to the environment and health of individuals in the vicinity of the real-time-system. The possibility of catastrophic consequences defines such systems as safety-critical. Real-time systems in a safety-critical environment have demanding requirements for certification, especially when it comes to safety. Due to their deterministic nature, time-triggered systems are an attractive option as to how to implement a real-time system in a safety-critical environment. Time-triggered systems provide temporal isolation of tasks running in their system and work with an offline constructed schedule. Scheduling constraints are solved before the system is deployed and thus the points of execution of all tasks in the systems are known beforehand. This behaviour provides deterministic execution, eases the detection of abnormal behaviour and aids the certification process. However, the very properties of real-time systems as well as time-triggered systems also provide an attack surface for attackers.

Threats to security also pose threats to the safety of a system. One cannot be provided without the other, as especially attacks on passenger vehicles like trains, airplanes and cars have shown recently. We analysed several vulnerabilities of real-time systems in general and time-triggered systems in particular. The timeliness guarantees real-time systems provide are a vulnerability in themselves. If attackers manage to disrupt timely behaviour of a real-time system, they defeat its very purpose. Additionally, any security measure implemented in software requires processor time which other tasks in the system may need to meet their respective deadlines. This may lead to situations where system designers sacrifice certain security measures for an improved performance or higher quality of service.

Another vulnerability lies in the derivation and optimization of worst-case execution time. Without assumptions on the system state, the bounds derived for the worst-case are quite pessimistic. Therefore, a body of works exists to reduce this pessimism. These works often make assumptions based on the observed system behaviour. However, a system under attack may not behave as intended or observed. This deviation might lead to a violation of the worst-case derived by observing only intended behaviour. Safe bounds on execution time therefore need to be derived under the anticipation of malicious behaviour. Lastly and specific to time-triggered systems, deterministic task execution aids an attacker in pinpointing the points in time where an attack is most disruptive. Schedules consist of few bytes of information and can be inferred through various means. Knowledge of the

schedule enables an attacker to carry out an attack at the most effective time and remain stealthy otherwise. We presented several strategies to mitigate directed attacks based on timing inference on single-core as well as multicore systems.

In order for the considered attack to succeed, the attacker needs to know the points in time when the victim task executes. Our mitigation strategies randomly change these execution times while adhering to timing constraints. Instead of the static behaviour of a single offline constructed schedule characteristic of time-triggered systems, the attacker observes changing points in time when the victim executes. Predictions on these points in time are not possible anymore. First, we introduced online slot-level randomization, which impedes predictions about the schedule by selecting the next job at the beginning of each slot at random. We employ concepts of slot shifting to allow randomization of a time-triggered schedule without violating deadlines. Secondly, we proposed an online selection of offline constructed schedules for the mitigation of directed attacks. During runtime, this mitigation strategy randomly selects a schedule from a set of precomputed schedules at the end of each hyperperiod. Both approaches do not need additional hardware as they can be implemented as scheduling algorithms.

We implemented and evaluated the mitigation strategies as scheduling algorithms on an open-source real-time operating system, FreeRTOS. The hardware platform chosen is used in avionics research and we used a task set for an avionics flight controller for evaluation. Both mitigation strategies prove to have low runtime overhead and memory cost and thus are a feasible choice for time-triggered real-time systems. In the worst case, the offline schedule-diversification algorithm presents a 30 %, the online slot-level randomization algorithm a 71 % runtime overhead increase over static time-triggered scheduling. Randomizing task execution online on slot-level entails a higher scheduling runtime overhead, but consumes less memory than storing a set of schedules constructed offline. Online slot-level randomization is also able to eventually explore all feasible schedules. However, highly constrained task sets or task sets with high utilization do not offer much possibility of randomization. The presented mitigation strategies are therefore a trade-off of an increase of either memory or scheduling overhead for an increase in security. Attackers can still try to launch undirected attacks, but they will be easier to detect this way.

For future work, we intend to investigate vulnerable worst-case execution time derivation and quantify the difference between bounds on execution time derived under intended behaviour in contrast to behaviour under attack. Furthermore, we intend to explore additional pitfalls and vulnerabilities of replicated multicore systems and develop mitigation strategies accordingly. Hardening against intrusion and isolating tasks such that even an hijacked task is not able to influence system behaviour might also mitigate the threat we consider for this work.

8.1 Final Remarks

Security is an ongoing battle between attackers finding new attack vectors and pathways into systems, and systems designers as well as hardware and software developers trying to protect and harden their systems against these attacks. New system designs may create new vulnerabilities, and creative attackers might also construct a previously unknown attack vector based upon already known vulnerabilities, attack countermeasures, or system properties. If a vulnerability or attack has been detected, made public and analysed, current and future systems are able to find ways to close the vulnerability or defend against the attack. However, if the attack vector or the vulnerability it exploits is not previously known, as it is the case with zero-days, defense and mitigation become much harder. This situation is aggravated by legal, governmental, financial or personal interests which may motivate individuals, companies or agencies to not publish their insights. Therefore, it is important to not only anticipate new attack vectors and thoroughly test the security of systems, during design as well as during deployment. Of paramount importance is also a cooperative culture, incentives for bughunts and penetrations tests, and legal protections for the disclosers of vulnerabilities and zero-days.

Security metrics are a method to check specific properties of the system with respect to security. However, they need to be chosen carefully when evaluating a system's security. We showed that schedule diversity as a metric helps to find schedules most distinct from each other, but that set of schedules might also be beneficial for an attacker. If the schedules are maximally diverse, each distinct schedule might be recognized early and thus the metric defeats its own purpose. Security metrics may thus be able to quantify a certain property of the system, but they might not be able to quantify overall system security.

I sincerely hope this thesis helps to strengthen system security. Security violations have become more common over time and the trend seems to keep increasing as we continue to add more and more computer systems to every aspect of our lives. I therefore foresee security to remain an important and relevant topic, in research as well as in everyday life. In the end, only a secure system is able to provide safe operation.

9 KURZFASSUNG

Die Zahl der Cyberangriffe steigt kontinuierlich und weitet sich auf jede Art von Computersystem aus. Echtzeitsysteme bieten zusätzliche Angriffsfläche, da sie nicht nur korrekt arbeiten, sondern ihre Ausgaben auch weder zu früh noch zu spät liefern müssen. Somit versagt ein Echtzeitsystem auch, wenn seine Ausgabe korrekt ist, jedoch nicht innerhalb einer vorgegebenen Zeitspanne erfolgt. Angreifer erhalten somit eine weitere Möglichkeit, für einen Systemausfall sorgen zu können.

Zeitgesteuerte Systeme sind eine Untergruppe von Echtzeitsystemen, die jedem Prozess ein bestimmtes Zeitfenster zur Abarbeitung in der zentralen Verarbeitungseinheit (CPU) zuweisen. Sie bieten somit nicht nur zeitliche Garantien, sondern auch eine vollständig vorhersagbare Abarbeitungsreihenfolge der Prozesse, die bereits vor dem Start des Systems bekannt ist. Diese Prozessreihenfolge wird in einem Ablaufplan, auch Schedule genannt, festgehalten, aus dem klar hervorgeht, wann welcher Prozess in der zentralen Verarbeitungseinheit abgearbeitet wird. Ein derart vorhersagbares zeitliches Verhalten ist in Bezug auf funktionale Sicherheit, Zuverlässigkeit und Systemzertifizierung wünschenswert. Allerdings birgt dieses Verhalten auch ein verschärftes Risiko für einen Cyberangriff. Angreifer können das vorhersagbare Systemverhalten ausnutzen, um ihre Angriffe gezielt und heimlich durchzuführen. Mit der Möglichkeit, den genauen Zeitpunkt der Prozessausführung vorhersagen zu können, können sie ihre Angriffe fein abstimmen, um ihr Ziel zu treffen. Da zeitgesteuerte Echtzeitsysteme häufig für sicherheitskritische Anwendungen verwendet werden, kann ein solcher Angriff fatale Folgen nach sich ziehen.

Um den Zeitpunkt der Prozessausführung vorhersagen zu können, müssen Angreifer den Ablaufplan des Computersystems kennen. Hierzu sind Seitenkanäle eine prominente Möglichkeit, um über nicht vorgesehene Kommunikationskanäle an (anderweitig unzugängliche) Informationen zu gelangen. Sicherheitsvorfälle wie Spectre [37] und Meltdown [45] haben gezeigt, dass Seitenkanäle in Computersystemen durchaus weit verbreitet sind. Zudem bestehen die Ablaufpläne solcher zeitgesteuerter Echtzeitsysteme nur aus wenigen Bytes an Information, weswegen selbst Seitenkanäle mit sehr geringer Bandbreite zum Abgreifen der Schedule ausreichen. So kann der Ablaufplan eines Computersystems sogar dann in Erfahrung gebracht werden, wenn er nicht öffentlich zugänglich ist. Daraus resultierende, gezielte Angriffe zu günstigen Gelegenheiten haben eine höhere Erfolgswahrscheinlichkeit und senken zugleich die Wahrscheinlichkeit einer Entdeckung des Angriffs durch Sicherheitsmaßnahmen im Computersystem.

Um solche gezielten Angriffe zu vereiteln, stellen wir zwei Algorithmen für Prozess-Scheduler eines Echtzeitsystems vor, die Zufallsgeneratoren einsetzen, um Vorhersagen

über die Ausführungszeitpunkte der Prozesse verhindern. Zufällig gewählte Ausführungszeitpunkte können jedoch dazu führen, dass Prozesse ihr Zeitfenster zur Abarbeitung nicht mehr einhalten können. Deshalb beachten unsere Algorithmen trotz der Randomisierung die zeitlichen Anforderungen der einzelnen Prozesse und stellen sicher, dass die zeitlichen Anforderungen eingehalten werden. Darüber hinaus beleuchten wir in dieser Dissertation auch weitere Schwachstellen von Echtzeitsystemen, die Angreifer ausnutzen könnten. Wir betrachten insbesondere Schwachstellen, die durch die grundlegenden Eigenschaften von Echtzeitsystemen entstehen. Der Beitrag dieser Dissertation lässt sich in drei Themengebiete unterteilen, auf die wir in den folgenden Abschnitten eingehen.

9.1 Schwachstellen von Echtzeit- und Zeitgesteuerten Systemen

Zunächst untersuchen wir Echtzeit- und zeitgesteuerte Systeme auf Schwachstellen, die sich aus ihren jeweiligen Anforderungen und Systemeigenschaften ergeben. Einerseits teilen sie sich die "klassischen" Sicherheitsprobleme anderer Computersysteme, doch aufgrund zusätzlicher Anforderungen an ihr Zeitverhalten und die Vorhersagbarkeit der Prozessausführung ergeben sich neue Angriffsflächen. Generell bietet jede zusätzliche Anforderung, der ein System entsprechen muss, Potential für eine solche Angriffsfläche. Wir analysieren Schwachstellen, die sich aus:

a) Anforderungen an das Zeitverhalten,

b) der Bestimmung der maximalen Abarbeitungszeit eines Prozesses basierend auf dem Systemverhalten, und

c) der vorhersagbaren Abarbeitungsreihenfolge der Prozesse ergeben.

Wir fokussieren uns auf die Schwachstelle, die sich aus der Vorhersagbarkeit der Prozessreihenfolge ergibt, präzisieren, wie ein Angreifer diese Schwachstelle für sich nutzen kann, und entwickeln Strategien gegen solche Angriffe.

9.2 Entwicklung von Scheduling-Algorithmen zur Vereitelung Gezielter Angriffe

Wir entwickeln Algorithmen für den Prozess-Scheduler, um gezielte Angriffe zu vereiteln, die sich die vorhersagbare Abarbeitungsreihenfolge von Prozessen in zeitgesteuerten Echtzeitsystemen zunutze machen. Für Systeme mit einem einzelnen Prozessorkern präsentieren wir zwei verschiedene Algorithmen:

a) Online Slot-level Randomization (OSR) ist ein Algorithmus, der vor der Laufzeit des Systems jeden Prozess hinsichtlich seiner zeitlichen Anforderungen untersucht. Er analysiert, innerhalb welcher Zeitfenster die verschiedenen Prozesse ausgeführt werden müssen und welche maximalen Abarbeitungszeiten sie besitzen. Anhand

dieser Information legt er ein Budget für verschiedene Zeitintervalle fest, in denen der Zufallsgenerator frei wählen kann, welcher Prozess als nächstes abgearbeitet wird. Jede Abarbeitung eines Prozesses beeinflusst das noch vorhandene Budget, abhängig von dem Zeitfenster des abgearbeiteten Prozesses. Wird das Budget überschritten, kann nur noch innerhalb einer bestimmten Teilmenge der Prozesse eine zufällige Wahl getroffen werden. Dadurch wird einerseits das Einhalten der zeitlichen Anforderungen garantiert und andererseits der Ausführungszeitpunkt der einzelnen Prozesse randomisiert.

b) Offline-Schedule Diversification (OSD) ist der zweite Algorithmus, der gezielte Angriffe vereitelt. Statt zur Laufzeit zufällige Entscheidungen zu treffen, wird für diesen Algorithmus vor der Laufzeit mehr als ein Ablaufplan für das zeitgesteuerte System erstellt. Jeder dieser Ablaufpläne berücksichtigt die Anforderungen an das Zeitverhalten und hält sie ein. Nach der Abarbeitung eines Ablaufplans wird zufällig ein anderer Plan aus der Menge aller Ablaufpläne gewählt, sodass sich das zeitliche Verhalten des Systems ändert. Durch eine entsprechend große Menge an Ablaufplänen und eine zufällige Auswahl wird verhindert, dass ein Angreifer Vorhersagen treffen kann.

Beide Scheduling-Algorithmen lassen sich auch auf Mehrkernprozessor-Systemen verwenden, solange eine strikte Zuweisung von Prozessen zu Prozessorkernen existiert. Zudem erlauben Mehrkernprozessor-Systeme eine hybride Strategie, bei der die Online-Randomisierung mit der Offline-Diversifikation kombiniert wird. Dadurch verhindern wir, dass Prozessorkerne den selben Prozess zur selben Zeit ausführen können, um einen gleichzeitigen Angriff auf den selben Prozess auf mehreren Kernen zu unterbinden.

9.3 Implementierung und Evaluierung der Algorithmen

Nach der Konzeption der beiden Scheduling-Algorithmen prüfen wir ihre Einsatztauglichkeit. Zuerst schätzen wir die Laufzeiten mit Messungen einer Simulation ab, um eine grobe Einschätzung zu erhalten. Im Anschluss implementieren wir sowohl Online Slot-level Randomization (OSR) als auch Offline-Schedule Diversification (OSD) als Scheduling-Algorithmen auf dem Echtzeit-Betriebssystem FreeRTOS. Neben diesen beiden Scheduling-Strategien implementieren wir auch den klassischen Scheduling-Algorithmus zeitgesteuerter Echtzeitsysteme, um unsere Ergebnisse vergleichen zu können. Wir wählen eine Fallstudie namens ROSACE aus dem sicherheitskritischen Bereich der Luftfahrt, die typische Prozesse eines Flugzeugkontrollsystems und deren Parameter enthält, um unsere Algorithmen zu evaluieren. Mithilfe dieser Fallstudie bestimmen wir die Laufzeiten, den Speicherbedarf, die Anzahl zulässiger Ablaufpläne und die Granularität des Determinismus, den wir auf Ebene der Zeitspannen der jeweiligen Prozesse noch erhalten können.

Wir stellen fest, dass unsere Scheduling-Algorithmen sowohl geringe Laufzeiten als auch realisierbaren Speicherbedarf besitzen. Im Vergleich zum klassischen Algorithmus zeitgesteuerter Systeme verlängert sich die Laufzeit von OSR im schlimmsten Fall um 71 Prozent und der Speicherbedarf beträgt das Viereinhalbfache. Im Fall von OSD stellen

wir eine Laufzeitverlängerung von maximal 30 Prozent fest. Der Speicherbedarf wächst hierbei linear mit der Anzahl an Ablaufplänen an, wobei bereits 1 MB an Speicher für etwa 21000 verschiedene Ablaufpläne ausreichen kann. Während OSR längere Laufzeiten besitzt, ist der Speicherbedarf jedoch geringer und der Algorithmus zudem in der Lage, mit der Zeit alle möglichen Ablaufpläne zu durchlaufen. Somit empfehlen wir OSR, sofern die längeren Laufzeiten hinnehmbar sind. Bei OSD muss stets der Speicherbedarf für die Ablaufpläne und somit auch deren Anzahl abgewogen werden, doch dafür besitzt es geringere Laufzeiten als OSR.

Die vorgestellten Algorithmen bedeuten einen Kompromiss zwischen längeren Laufzeiten und erhöhtem Speicherbedarf zur Vereitlung gezielter Angriff einzugehen. Angreifer können zwar noch ungezielte Angriffe starten, doch diese lassen sich leichter entdecken und nach einer Entdeckung können geeignete Gegenmaßnahmen ergriffen werden. Mit dieser Arbeit möchten wir einen Beitrag zur Verkleinerung der Angriffsfläche von Echtzeit- und zeitgesteuerten Systemen leisten, um die Entwicklung von sicheren und zuverlässigen Systemen zu unterstützen.

七転び八起き

Fall down seven times, get up eight.

LIST OF FIGURES

2.1 Exemplary task τ_1 with period T and jobs $\tau_{1,0}$, $\tau_{1,1}$, $\tau_{1,2}$ 10
2.2 Simple process state model 10
2.3 Exemplary task with explicit deadline 11
2.4 Exemplary task with implicit deadline 11
2.5 FreeRTOS logo [8] 15
2.6 Exemplary uptime and downtime during system lifetime 16
2.7 Downtime per year for several example components [67] 17
2.8 Fault, error, failure with propagation 18
2.9 Systems tells value of x while experiencing crash and Byzantine fault, respectively 18
2.10 Majority vote of three task replicas which masks the faulty output of replica 2 19
2.11 Taxonomy of research on real-time systems and security according to [32]. Purple frames indicate the topics this thesis addresses. References in the figure refer to [32]. 23
2.12 Domains of moving target defense compared against a typical compute stack [55]. 26

3.1 Meltdown and Spectre logos [25] 31

5.1 Job set and corresponding capacity intervals 42
5.2 Left: spare capacities of I_1, I_2 and I_3 over time, Right: randomized schedule 44

6.1 Attacker and victim task on the same core c_i 52
6.2 Attacker on core c_i attacks victim task running in parallel on another core c_j 52
6.3 Attacker on core c_i attacks victim task running next on another core c_j . . 53

7.1 Longitudinal flight controller design 58
7.2 MicroZed 7020 Board 61
7.3 Architectural overview of a FreeRTOS project 64
7.4 Handling of a timer interrupt in FreeRTOS 66
7.5 Changes to FreeRTOS functionality 68
7.6 Time-triggered scheduler flow chart 71
7.7 Offline schedule-diversification scheduler flow chart 72
7.8 Online slot-level randomization scheduler flow chart 73
7.9 Architectural overview of changes to FreeRTOS functionality 74

7.10 Scheduling runtime overhead measurement overview including significant events, timestamps and FreeRTOS API calls 77
7.11 Visualization of scheduling runtime measurements (in μs) as box plots for online slot-level randomization (OSR), offline schedule diversification (OSD), and time-triggered scheduling (TT) 80
7.12 Visualization of API runtime measurements (in μs) as box plots 82
7.13 Different levels of determinism: top shows slot level, bottom execution window level 89

LIST OF TABLES

2.1 Different requirements for real-time and non-real-time systems 9

3.1 Example of an offline constructed schedule 35

7.1 Flight controller task set[56] 58
7.2 Execution windows in terms of slots for MPARM 59
7.3 Minimum and maximum runtime overhead per slot for single core execution in ns [61] 60
7.4 Minimum and maximum runtime overhead for single core execution in ns [61] 60
7.5 Execution windows in terms of slots for FreeRTOS 76
7.6 Best, mean and worst case scheduling runtime overhead measurements of ROSACE task set scheduled with online slot-level randomization, offline schedule diversification, and time-triggered scheduling 79
7.7 Best, mean and worst case runtime overhead measurements of different parts of the online slot-level randomization algorithm 81
7.8 Best, mean and worst case runtime overhead of measurement function parts, measured in clock cycles 83
7.9 Exemplary time-triggered schedule 84
7.10 Memory requirement of the task control block extension, where M is the number of capacity intervals 85
7.11 Memory requirement of capacity interval data structure, where N is the number of tasks belonging to the capacity interval 86

BIBLIOGRAPHY

[1] H. Aamir. *Bug in Mercedes-Benz app leaked car owners' data to other users*. 2019. URL: https://www.techspot.com/news/82420-bug-mercedes-benz-app-leaked-car-owners-data.html.

[2] D. Agrawal, B. Archambeault, et al. "The EM Side-Channel(s)". In: *4th International Workshop on Cryptographic Hardware and Embedded Systems*. CHES. 2002. DOI: 10.1007/3-540-36400-5_4.

[3] Amazon Web Services. *The FreeRTOS™ Reference Manual - API Functions and Configuration Options*. English. Version 10.0.0 issue 1. Amazon.com, Inc. or its affiliates. 304 pp.

[4] ARM. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. English. Version ARM DDI 0406C.d (ID040418). ARM.

[5] ARM. *Cortex™ -A9, Revision: r0p1, Technical Reference Manual*. English. Version ARM DDI 0388B. ARM. 362 pp.

[6] A. Avizienis, J. C. Laprie, et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.

[7] A. K. Bangalore and A. K. Sood. "Securing Web Servers Using Self Cleansing Intrusion Tolerance (SCIT)". In: *2009 Second International Conference on Dependability*. 2009, pp. 60–65. DOI: 10.1109/DEPEND.2009.15.

[8] R. Barry. *FreeRTOS™ Project*. Amazon Web Services. URL: https://www.freertos.org/.

[9] M. G. Bechtel and H. Yun. "Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention". In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019. DOI: 10.1109/RTAS.2019.00037.

[10] P. K. Boucher, R. K. Clark, et al. "Toward a Multilevel-Secure, Best-Effort Real-Time Scheduler". In: *Dependable Computing for Critical Applications 4*. Ed. by F. Cristian, G. Le Lann, et al. Vienna: Springer Vienna, 1995, pp. 49–68. ISBN: 978-3-7091-9396-9. DOI: 10.1007/978-3-7091-9396-9_8.

[11] L. Brandao and A. Bessani. "On the Reliability and Availability of Systems Tolerant to Stealth Intrusion". In: *5th Latin-American Symposium on Dependable Computing (LADC'11)*. Brazil, Apr. 2011. DOI: 10.1109/LADC.2011.27.

[12] L. Brandao and A. Bessani. “On the Reliability and Availability of Replicated and Rejuvenating Systems under Stealth Attacks and Intrusions”. In: *Journal of the Brazilian Computer Society* 18 (Mar. 2012), pp. 61–80. DOI: 10.1007/s13173-012-0062-x.

[13] O. Burkacky, J. Deichmann, et al. *Cybersecurity in automotive - Mastering the challenge*. 2020. URL: https://www.mckinsey.com/~/media/mckinsey/industries/automotive%20and%20assembly/our%20insights/cybersecurity%20in%20automotive%20mastering%20the%20challenge/cybersecurity-in-automotive-mastering-the-challenge.pdf.

[14] G. C. Buttazzo. *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications, Third Edition*. Springer, 2011.

[15] C. Cadar, P. Akritidis, et al. *Data randomization, Technical Report TR-2008-120*. Tech. rep. Microsoft Research, 2008.

[16] C.-Y. Chen, S. Mohan, et al. “A Novel Side-Channel in Real-Time Schedulers”. In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019, pp. 90–102. DOI: 10.1109/RTAS.2019.00016.

[17] J. Chen, T. Kloda, et al. “SchedGuard: Protecting against Schedule Leaks Using Linux Containers”. In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2021, pp. 14–26. DOI: 10.1109/RTAS52030.2021.00010.

[18] J. Chen, T. Kloda, et al. “SchedGuard++: Protecting against Schedule Leaks Using Linux Containers on Multi-Core Processors”. In: *ACM Trans. Cyber-Phys. Syst.* 7.1 (2023). ISSN: 2378-962X. DOI: 10.1145/3565974. URL: https://doi.org/10.1145/3565974.

[19] X. Chen, J. Feng, et al. “Application of software watchdog as a dependability software service for automotive safety relevant systems”. In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2007. DOI: 10.1109/DSN.2007.14.

[20] T. Claburn and K. Hall. *‘WHAT THE F*CK IS GOING ON?’ Linus Torvalds explodes at Intel spinning Spectre fix as a security feature*. 2018. URL: https://www.theregister.com/2018/01/22/intel_spectre_fix_linux/.

[21] I. B. M. Corporation. *SATURN V LAUNCH VEHICLE DIGITAL COMPUTER, Laboratory Maintenance Instructions, Volume 1 General Description and Theory*. Tech. rep. IBM, 1964.

[22] S. S. Craciunas and R. S. Oliver. “SMT-based Task- and Network-level Static Schedule Generation for Time-Triggered Networked Systems”. In: *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*. RTNS ’14. Versaille, France: ACM, 2014, 45:45–45:54. ISBN: 978-1-4503-2727-5. DOI: 10.1145/2659787.2659812.

[23] L. Davis. “Modifications of the optimal velocity traffic model to include delay due to driver reaction time”. In: *Physica A: Statistical Mechanics and its Applications* 319 (2003), pp. 557–567. ISSN: 0378-4371. DOI: https://doi.org/10.1016/S0378-4371(02)01457-7. URL: https://www.sciencedirect.com/science/article/pii/S0378437102014577.

[24] DeepWiki. *FreeRTOS Architecture Overview*. URL: https://deepwiki.com/FreeRTOS/FreeRTOS-Kernel/4-portability-layer.

[25] N. Eibl. *Meltdown and Spectre logo*. CC0 via Wikimedia Commons.

[26] C. Ferdinand and R. Wilhelm. “Efficient and Precise Cache Behavior Prediction for Real-Time Systems”. In: *Real-Time Systems* 17.2 (Nov. 1999), pp. 131–181. ISSN: 1573-1383. DOI: 10.1023/A:1008186323068.

[27] G. Fohler. “Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems”. In: *Proceedings 16th IEEE Real-Time Systems Symposium*. Dec. 1995, pp. 152–161. DOI: 10.1109/REAL.1995.495205.

[28] G. Fohler. *Advances in Real-Time Systems, Chapter Predictably Flexible Real-time Scheduling*. SPRINGER, 2012.

[29] G. Fohler, G. Gala, et al. “Evaluation of DREAMS resource management solutions on a mixed-critical demonstrator”. In: *ERTS 2018*. 2018.

[30] M. Franz. “E unibus pluram: massive-scale software diversity as a defense mechanism”. In: *Proceedings of the 2010 New Security Paradigms Workshop*. NSPW ’10. Concord, Massachusetts, USA: Association for Computing Machinery, 2010, pp. 7–16. ISBN: 9781450304153. DOI: 10.1145/1900546.1900550. URL: https://doi.org/10.1145/1900546.1900550.

[31] A. Greenberg. *Hackers Remotely Kill a Jeep on the Highway—With Me in It*. 2015. URL: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[32] M. Hasan, A. Kashinath, et al. “SoK: Security in Real-Time Systems”. In: *ACM Comput. Surv.* 56.9 (Apr. 2024). ISSN: 0360-0300. DOI: 10.1145/3649499. URL: https://doi.org/10.1145/3649499.

[33] W. M. Hu. “Lattice scheduling and covert channels”. In: *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. May 1992, pp. 52–61. DOI: 10.1109/RISP.1992.213271.

[34] B. K. Huynh, L. Ju, et al. “Scope-Aware Data Cache Analysis for WCET Estimation”. In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. Apr. 2011, pp. 203–212. DOI: 10.1109/RTAS.2011.27.

[35] K. Jiang, P. Eles, et al. “SPARTA: A scheduling policy for thwarting differential power analysis attacks”. In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. schedule randomization example for EDF, randomize leakage points of AES cryptography against differential power analysis attacks. 2016, pp. 667–672. DOI: 10.1109/ASPDAC.2016.7428088.

[36] I. King, J. Kahn, et al. *'It Can't Be True.' Inside the Chip Industry's Meltdown*. 2018. URL: https://www.bloomberg.com/news/articles/2018-01-08/-it-can-t-be-true-inside-the-semiconductor-industry-s-meltdown.

[37] P. Kocher, J. Horn, et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[38] H. Kopetz. "Sparse time versus dense time in distributed real-time systems". In: *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*. June 1992, pp. 460–467. DOI: 10.1109/ICDCS.1992.235008.

[39] H. Kopetz. *Real-Time System - Design Principles for Distributed Embedded Applications, Second Edition*. Springer, 2011.

[40] K. Krüger, G. Fohler, et al. "Improving Security for Time-Triggered Real-Time Systems with Task Replication". In: *24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2018. DOI: 10.1109/RTCSA.2018.00036.

[41] K. Krüger, M. Völp, et al. "Improving Security for Time-Triggered Real-Time Systems against Timing Inference Based Attacks by Schedule Obfuscation". In: *Work-in-Progress Proceedings of the 29th Euromicro Conference on Real-Time Systems*. ECRTS. 2017.

[42] K. Krüger, M. Völp, et al. "Vulnerability Analysis and Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Systems". In: *Euromicro Conference on Real-Time Systems*. ECRTS. 2018. DOI: 10.4230/LIPIcs.ECRTS.2018.22.

[43] K. Krüger, N. Vreman, et al. "Randomization as Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Real-Time Systems with Task Replication". In: *Leibniz Transactions on Embedded Systems* 7.1 (Aug. 2021), 01:1–01:29. DOI: 10.4230/LITES.7.1.1. URL: https://ojs.dagstuhl.de/index.php/lites/article/view/lites-v007-i001-a001.

[44] J. Liedtke, H. Hartig, et al. "OS-controlled cache predictability for real-time systems". In: *Proceedings Third IEEE Real-Time Technology and Applications Symposium*. June 1997, pp. 213–224. DOI: 10.1109/RTTAS.1997.601360.

[45] M. Lipp, M. Schwarz, et al. "Meltdown". In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01207.

[46] S. Liu, N. Guan, et al. "Leaking Your Engine Speed by Spectrum Analysis of Real-Time Scheduling Sequences". In: *Journal of Systems Architecture* (2019). ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2019.01.004.

[47] S. Liu and W. Yi. "Task Parameters Analysis in Schedule-Based Timing Side-Channel Attack". In: *IEEE Access* 8 (2020), pp. 157103–157115. DOI: 10.1109/ACCESS.2020.3019323.

[48] S. Lu, M. Seo, et al. “Timing-based anomaly detection in embedded systems”. In: *20th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2015. DOI: 10.1109/ASPDAC.2015.7059110.

[49] E. Mateos and C. Gebotys. “A New Correlation Frequency Analysis of the Side Channel”. In: *Workshop on Embedded Systems Security*. 2010. DOI: 10.1145/1873548.1873552.

[50] C. Miller and C. Valasek. “Remote Exploitation of an Unaltered Passenger Vehicle”. In: *BlackHat USA 2015*. 2015. URL: https://www.blackhat.com/us-15/briefings.html#remote-exploitation-of-an-unaltered-passenger-vehicle.

[51] R. Mitchell and I.-R. Chen. “A survey of intrusion detection techniques for cyber-physical systems”. In: *ACM Computing Surveys (CSUR)* 46.4 (2014). DOI: 10.1145/2542049.

[52] S. Mohan, M.-K. Yoon, et al. “Integrating security constraints into fixed priority real-time schedulers”. In: *Real-Time Systems* (2016), pp. 1–31. DOI: 10.1007/s11241-016-9252-5.

[53] M. Mushtaq, M. A. Mukhtar, et al. “Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA”. In: *Information Systems* 92 (2020), p. 101524. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2020.101524. URL: https://www.sciencedirect.com/science/article/pii/S0306437920300338.

[54] M. Nasri, T. Chantem, et al. “On the Pitfalls and Vulnerabilities of Schedule Randomization against Schedule-Based Attacks”. In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019.

[55] H. Okhravi. “Moving Target Defense”. In: *Encyclopedia of Cryptography, Security and Privacy*. Ed. by S. Jajodia, P. Samarati, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, pp. 1–4. ISBN: 978-3-642-27739-9. DOI: 10.1007/978-3-642-27739-9_1770-1. URL: https://doi.org/10.1007/978-3-642-27739-9_1770-1.

[56] C. Pagetti, D. Saussié, et al. “The ROSACE case study: From Simulink specification to multi/many-core execution”. In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2014, pp. 309–318. DOI: 10.1109/RTAS.2014.6926012.

[57] T. Popp, S. Mangard, et al. “Power analysis attacks and countermeasures”. In: *IEEE Design & test of Computers* 24.6 (2007). DOI: 10.1109/MDT.2007.200.

[58] S. Ravi, A. Raghunathan, et al. “Security in Embedded Systems: Design Challenges”. In: *ACM Trans. Embed. Comput. Syst.* 3.3 (Aug. 2004), pp. 461–491. ISSN: 1539-9087. DOI: 10.1145/1015047.1015049. URL: http://doi.acm.org/10.1145/1015047.1015049.

[59] Richard Barry and The FreeRTOS Team. *Mastering the FreeRTOS™ Real Time Kernel - A Hands-On Tutorial Guide*. English. Version 1.1.0. Amazon.com, Inc. or its affiliates. 405 pp.

[60] A. Samaddar, A. Easwaran, et al. "A schedule randomization policy to mitigate timing attacks in WirelessHART networks". In: *Real-Time Systems* (Aug. 2020). schedule randomization method for wireless networks. DOI: 10.1007/s11241-020-09354-z. URL: https://doi.org/10.1007/s11241-020-09354-z.

[61] S. Schorr. "Adaptive Real-Time Scheduling and Resource Management on Multicore Architectures". PhD thesis. Technical University of Kaiserslautern, Mar. 2015. URL: https://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/4008.

[62] S. Sibi Chakkaravarthy, D. Sangeetha, et al. "A Survey on malware analysis and mitigation techniques". In: *Computer Science Review* 32 (2019), pp. 1–23. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2019.01.002. URL: https://www.sciencedirect.com/science/article/pii/S1574013718301114.

[63] F. Skopik, A. Treytl, et al. "Towards Secure Time-Triggered Systems". In: *Computer Safety, Reliability, and Security: SAFECOMP 2012 Workshops: Sassur, ASCoMS, DESEC4LCCI, ERCIM/EWICS, IWDE, Magdeburg, Germany, September 25-28, 2012. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 365–372. ISBN: 978-3-642-33675-1. DOI: 10.1007/978-3-642-33675-1_33.

[64] J. Son and J. Alves-Foss. "Covert timing channel capacity of rate monotonic real-time scheduling algorithm in MLS systems". In: *Communication, Network, and Information Security*. 2006.

[65] R. Spreitzer, V. Moonsamy, et al. "Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices". In: *IEEE Communications Surveys and Tutorials* 20.1 (2018). DOI: 10.1109/COMST.2017.2779824.

[66] W. Stallings. *Operating Systems - Internals and Design Principles*. Pearson Education Limited, 2018.

[67] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2004.

[68] M. Völp, B. Engel, et al. "On confidentiality-preserving real-time locking protocols". In: *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2013. DOI: 10.1109/RTAS.2013.6531088.

[69] M. Völp, C.-J. Hamann, et al. "Avoiding Timing Channels in Fixed-priority Schedulers". In: *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*. ASIACCS '08. Tokyo, Japan: ACM, 2008, pp. 44–55. ISBN: 978-1-59593-979-1. DOI: 10.1145/1368310.1368320.

[70] N. Vreman, R. Pates, et al. "Minimizing Side-Channel Attack Vulnerability via Schedule Randomization". In: *58th IEEE Conference on Decision and Control (CDC)*. Dec. 2019. DOI: 10.1109/CDC40024.2019.9030144.

[71] A. Wasicek, C. El-Salloum, et al. “Authentication in Time-Triggered Systems Using Time-Delayed Release of Keys”. In: *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. Mar. 2011, pp. 31–39. DOI: 10.1109/ISORC.2011.14.

[72] A. R. Wasicek. “Security in Time-Triggered Systems”. PhD thesis. Technische Universität Wien, 2011.

[73] R. Wilhelm, J. Engblom, et al. “The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools”. In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389. URL: http://doi.acm.org/10.1145/1347375.1347389.

[74] M. Yoon, S. Mohan, et al. “TaskShuffler: A Schedule Randomization Protocol for Obfuscation against Timing Inference Attacks in Real-Time Systems”. In: *IEEE Real-Time and Embedded Technology and Applications Symposium*. RTAS. 2016. DOI: 10.1109/RTAS.2016.7461362.

[75] H. Yun, R. Mancuso, et al. “PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms”. In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2014, pp. 155–166. DOI: 10.1109/RTAS.2014.6925999.

[76] C. Zimmer, B. Bhat, et al. “Time-based intrusion detection in cyber-physical systems”. In: *1st ACM/IEEE International Conference on Cyber-Physical Systems*. 2010. DOI: 10.1145/1795194.1795210.

LEBENSLAUF

Name: Kristin Krüger
Geburtsort: Schwetzingen, Deutschland

WERDEGANG

09/2015 – 08/2026	:	Wissenschaftliche Mitarbeiterin an der Rheinland-Pfälzischen Technischen Universität Kaiserslautern-Landau, Deutschland
10/2009 – 08/2015	:	Studium an der TU Kaiserslautern Diplom Informationstechnik (Computer Engineering) Schwerpunkt: Embedded Systems, Note: 1,4

LEHRE

2019 – 2026	:	Administration Vorlesung Assemblerprogrammierung Leitung Übung Assemblerprogrammierung SS 21 und SS 22 Übung 1. Platz Vorlesungsumfrage SS 26 Übung 2. Platz Vorlesungsumfrage
2016 – 2018	:	Administration Vorlesung Real-Time Systems Leitung Übung Real-Time Systems

EHRENAMT

2019 – 2026	:	Prüfungsausschuss Elektro- und Informationstechnik
2017 – 2025	:	Fachbereichsrat Elektro- und Informationstechnik
2009 – 2015	:	Fachschaftsrat Elektro- und Informationstechnik