Randomization as Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Real-Time Systems with Task Replication

Kristin Krüger
Department of Electrical and Computer Engineering, Technische Universität Kaiserslautern, Kaiserlautern, Germany

Nils Vreman
Department of Automatic Control, Lund University, Lund, Sweden

Richard Pates
Department of Automatic Control, Lund University, Lund, Sweden

Martina Maggio
Department of Automatic Control, Lund University, Lund, Sweden
Department of Computer Science, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Marcus Völp
SnT – Université du Luxembourg, Esch-sur-Alzette, Luxembourg

Gerhard Fohler
Department of Electrical and Computer Engineering, Technische Universität Kaiserslautern, Kaiserslautern, Germany

Abstract

Time-triggered real-time systems achieve deterministic behavior using schedules that are constructed offline, based on scheduling constraints. Their deterministic behavior makes time-triggered systems suitable for usage in safety-critical environments, like avionics. However, this determinism also allows attackers to fine-tune attacks that can be carried out after studying the behavior of the system through side channels, targeting safety-critical victim tasks. Replication – i.e., the execution of task variants across different cores – is inherently able to tolerate both accidental and malicious faults (i.e. attacks) as long as these faults are independent of one another. Yet, targeted attacks on the timing behavior of tasks which utilize information gained about the system behavior violate the fault independence assumption fault tolerance is based on. This violation may give attackers the opportunity to compromise all replicas simultaneously, in particular if they can mount the attack from already compromised components. In this paper, we analyze vulnerabilities of time-triggered systems, focusing on safety-certified multicore real-time systems. We introduce two runtime mitigation strategies to withstand directed timing inference based attacks: (i) schedule randomization at slot level, and (ii) randomization within a set of offline constructed schedules. We evaluate these mitigation strategies with synthetic experiments and a real case study to show their effectiveness and practicality.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Security and privacy → Operating systems security; Software and its engineering → Scheduling; Computer systems organization → Redundancy

Keywords and Phrases real-time systems, time-triggered systems, security

Digital Object Identifier 10.4230/LITES.7.1.1

Supplementary Material *Software*: `https://gitlab.control.lth.se/NilsVreman/rand-sched`

Funding This work is partially supported by: (i) the EC through H2020 grant 871259, ADMORPH, (ii) funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 432878494, and (iii) FNR by FNR-Core Jr. Project HyLIT – CS18/IS/12686210. Nils Vreman, Richard Pates, and Martina Maggio are members of the ELLIIT stategic research area at Lund University.

Acknowledgements We want to thank the reviewers for their helpful comments which greatly improved this paper. We are also gratefully indebted to Florian Heilman, Gautam Gala, Luiz Maia and Alexandre Venito from TU Kaiserslautern for their comments on an earlier version of this paper. Their insights and expertise assisted research, however, any errors found are our own.

Received 2020-04-14 **Accepted** 2020-12-14 **Published** 2021-08-12

Editor Alan Burns and Steve Goddard

Special Issue Special Issue on Embedded System Security

1 Introduction

In the past, real-time systems security had been given little thought, primarily because systems were closed, ran on specialized hardware, and had limited access from the outside. This has changed in the past decade due to the growing need for connectivity, computing power, the shift from single to multi- and many-core architectures, and software-component reuse (e.g., from federated avionics architectures to Integrated Modular Avionics (IMA) [56]). These trends result in a general increase of complexity, in particular at real-time application level, with the entailed increase of the likelihood of vulnerabilities going undetected, even in well-tested software. Consequently, real-time system designers must now anticipate the risk of hackers trying to compromise components and being partially successful in this task. Security has become a primary concern during system design and deployment, not only to prevent unauthorized information disclosure, but also to prevent malicious exploitation of vulnerabilities [54]. This is especially true for systems in safety-critical environments, thus for the majority of real-time systems.

Real-time systems provide deterministic and dependable behavior, both in the value-domain and in the timeliness of responses. To achieve IEC 61508 or ISO 26262 compliance [19,21], industry employs time-triggered real-time systems which, in addition to guaranteeing that deadlines will be met, provide *determinism* in the sense that it is known which task executes at any given point in time. Unfortunately, the very essence of this determinism opens threat vectors that attackers can exploit to harm the system and the environment in which it operates. Aside from inheriting all classic security concerns related to providing correct, trustworthy results, real-time systems must also produce these results in time. Attackers may exploit this property by delaying the execution of individual tasks in targeted time-domain attacks. Moreover, the traditional means for tolerating accidental faults, for example active replication [10, 14, 20] or their counterparts for intentionally malicious faults [2, 28, 40], which require only a majority of components to work correctly, are not immune against time-domain attacks. Such attacks constitute a common mode fault against which replication does not protect, even if replicas are diversified (e.g., through n-version programming).

In general, attacks on the timing of tasks are more effective the more information adversaries obtain about the system and its schedule. Adversaries may fine-tune attacks to precise points in time and thus remain stealthy to evade detection and act when the attack is most effective. For such attacks, time-triggered systems appear the perfect target due to their deterministic schedules. On the other hand, this determinism seems to be the perfect protection as tasks can only execute during predefined time windows which do not overlap with other tasks execution. However, this perception hinges on the assumption of perfect isolation, which is brittle as long as tasks are analyzed only according to their specified behavior (e.g., when determining cache-related preemption delays) and as long as isolation remains imperfect. We therefore have to agree with Yoon et al. [58] in their conclusion that time-triggered scheduling is inherently vulnerable to timing inference based attacks.

In this article, we consolidate the findings of three independent works [25, 27, 53] of the authors into a comprehensive analysis of the vulnerabilities and countermeasures of time-triggered systems against timing interference based attacks. We investigate how adversaries can exploit vulnerabilities in an accomplice task to prepare and execute targeted attacks to singleton and replicated critical subsystems. Based on this analysis, we propose two basic strategies to mitigate attacks:

1. online randomization of the time-triggered schedule while preserving all timing constraints
2. random online selection of offline-prepared time-triggered schedules at hyperperiod boundaries.

These results, in particular the second, are formalized by constructing schedule sets of minimal size that achieve the highest possible upper-approximated entropy. Through this formalization,

we were able to conclude the effectiveness of online randomization in preventing adversaries from mounting targeted attacks. Moreover, we were able to conclude that a set of 100 offline computed schedules suffices for our second mitigation strategy. We have evaluated our approach using synthetic task sets and the ROSACE real-world case study.

In addition to already published results, this article extends our mathematical foundation from implicit to constrained deadline task sets (although the bound for the latter is not tight). The extension is purely theoretical, but shows that constrained deadlines reduce the achievable upper-approximated entropy (because they limit the amount of randomization that can be introduced in the system). The article also presents the results of an upper-approximated entropy limitation analysis for the ROSACE case study. This last result demonstrates that entropies close to the theoretical optimum can be achieved with realistic task sets that do not necessarily have the ideal parameter distribution for this optimum.

Section 2 introduces our system model. Section 3 describes our attacker model. We present two mitigation strategies in Section 4. In Section 5, we discuss inherent limitations of schedule diversity and randomization. We discuss attack vectors and mitigations strategies in a multicore system with task replication in Section 6. In Section 7, we evaluate our strategies using two sets of experiments: synthetic task sets and a real-world case study. We discuss our findings and explore limitations that are due to the implementation in embedded systems. Section 8 shows related work. Finally, we conclude our work in Section 9.

2 System model

We analyze a time-triggered real-time system implemented on a multicore platform, following a partitioned schedule. For each core, a local schedule is constructed offline and then adjusted online as discussed in Section 4. We assume the schedules have been validated. Precautions such as authenticated boot are in place to ensure that the validated schedule is correctly deployed to the real-time system. A real-time operating system is present, which we assume to be correct. That is, we do not consider bare metal real-time implementations. Safety-critical tasks are replicated and replicas are distributed across cores to benefit from improved resilience should one core fail. The implementation of these replicas follows an implementation diversification approach to increase system dependability. We assume the system is configured to be able to tolerate up to f faults of arbitrary kind simultaneously and focus in this work on what is required to tolerate time-domain attacks in addition. That is, our approach is equally applicable to systems prepared for tolerating accidental faults or maliciously induced (value-domain) faults and for tolerating crash as well as Byzantine faults. The replication degree, i.e. the number n of replicas required to achieve the mentioned fault tolerance, depends on this fault model, on the achieved system synchrony and on the kind of agreement, e.g. single value versus interval mid point. Again, our approach is prepared for any combination of the above. We therefore assume what the above models have in common, namely that no more than f replicas fail simultaneously and that replicas fail independently in the value-domain. We shall see that, without further precautions, a general fault independence requirement can no longer be maintained in the presence of time-domain attacks.

The scheduler on each core has access to a global time base typical for time-triggered systems. We consider global time adequately protected and divide it 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 purely time-triggered system without asynchronous event activation.

For each core c, we are given a task set $\mathcal{T}^c = \{\tau_1^c, \tau_2^c, \ldots, \tau_m^c\}$ of m^c periodic tasks. Each task τ_i^c is defined as the tuple $\tau_i^c = \{e_i^c, t_i^c, d_i^c\}$, where e_i^c is the worst-case execution time of the task[1] , t_i^c is the task activation period, and d_i^c is the task (relative) deadline. Throughout the rest of the paper, we assume that $d_i^c \leq t_i^c$, adopting the constrained deadline task model.

We define the time interval between task release and task deadline as this task's execution window. We denote with U^c the task set utilization for core c, i.e., $U^c = \sum_{i=1}^{m^c} e_i^c/t_i^c$. We assume that the task set is schedulable; ergo, there exists a resource distribution such that each task meets its corresponding deadline. We denote with ℓ^c the hyperperiod of the task set for core c, i.e., the least common multiple lcm$(\cdot)$ of the task periods, $\ell^c = \text{lcm}(t_1^c, \ldots, t_m^c)$.

We now look at a specific core c.[2] A schedule s^c for the task set in core c is a sequence of ℓ^c elements, that contains numbers in the set $\{0, 1, \ldots, m^c\}$, a number $j \in \{1, \ldots, m^c\}$ denotes the execution of task τ_j^c. Assigning a given task to an element means selecting which task is executed on core c for the corresponding time slot. Choosing $j = 0$ represents the execution of the idle task, meaning keeping the processor in the idle state. We denote the idle task with $\tau_0^c = \{e_0^c, t_0^c, d_0^c\}$ and we determine its characteristics based on the characteristics of the task set. In particular, $t_0^c = d_0^c = \ell^c$, and $e_0^c = \ell^c\ (1 - U^c)$.

Formally,

$$s^c = (s_1, s_2, \ldots, s_{\ell^c})\,;\, s_j^c \in \{0, 1, \ldots, m^c\}. \tag{1}$$

We denote with s_j^c the value of the element in position j, i.e., the task that is executed according to the schedule s^c in the j-th time unit. Given that the task set $\mathcal{T}^c$ is schedulable, we can safely assume that s^c respects the constraints that for each task and each activation, the schedule assigns to each task the required amount of execution time before the corresponding task deadline.

3 Threat Model and Vulnerability Analysis

In this section, we first describe our threat model, highlighting in particular the assumptions we make on the attacker and how he or she is constrained by time-triggered systems. After that, we analyze the vulnerabilities present in time-triggered systems.

3.1 Threat Model

We assume attackers are able to successfully infiltrate the system through undetected vulnerabilities. Less stringent evaluation requirements make non real-time tasks as well as not safety-critical tasks primary targets. In particular, we assume that critical tasks are sufficiently shielded against direct attacks which requires attackers to find a pathway through less critical tasks. Firewalls and gateways in autonomous vehicles and planes support this assumption. Even though we assume intrusion detection [8, 32, 35, 60], hardening mechanisms and other defenses against the common attack vectors (e.g. DoS attacks) are in place, we acknowledge that these techniques are imperfect and compromises may go undetected.

Of particular concern to us are stealthy attackers [6, 7] that 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

[1] To guarantee the correct execution of the task set, we ensure that a time equal to the worst-case execution time is assigned to each task in each of the execution periods (i.e., 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.

[2] In the following, when it is clear from the context that we talk about one specific core (and in particular in Section 5), we will drop the superscript c and use $\mathcal{T}$, τ_i, e_i, t_i, d_i, U, ℓ, s and s_j.

is most effective, e.g., immediately before a safety-critical victim task is run. Possible targets of such attacks in time-triggered systems are the low-level control loops. Destabilizing these components (e.g., by increasing the dead time or by introducing jitter in the control cycle) may provoke critical failure modes and thus result in a continuous denial of service [55], or worse, unsafe control decisions.

The timing information required for coordinating such a stealthy attack can be inferred via side channels constructed using shared resources like cache [3] or memory, or through covert timing channels, such as the scheduling-covert-channel described by Boucher et al. [5].

While there exist mitigation strategies for closing side channels (for example in the real-time context, the works of Völp et al. [52] or Mohan et al. [36] on fixed-priority schedulers), these methods are incomplete. Additionally, systematically closing all side channels typically entails significant performance overheads, e.g. when flushing caches prior to scheduling a lower classified task [17]. Meltdown [30] and Spectre [22] 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 high-end ARM and AMD processors. While real-time systems traditionally avoid such complex hardware, their future integration in real-time system-on-chip, e.g., for meeting the extended demand of autonomous driving functionalities, cannot be excluded.

We assume the real-time system features isolation mechanisms for enforcing the schedule of tasks and for limiting direct access to the memory of other tasks. Real-time operating systems (RTOS) that feature memory isolation support this assumption unless attackers are able to penetrate the operating system. For the purpose of this paper, we assume the deployed RTOS excludes the possibility of OS penetration.

One immediate consequence of this isolation assumption is that when the attacker has infiltrated the system, he or she is inherently constrained by properties of the system and its architecture for subsequent attacks on more critical tasks. In time-triggered systems, table-driven scheduling prevents influencing other tasks, e.g. 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 behavior of tasks and without having to revert to timing leak transformations as described for example by Völp et al. [52]. Additionally, messages are only accepted during a certain time window, i.e., if they are timely.

Operating system enforced schedules combined with the assumed impenetrability of the OS ensure that the attacker can neither directly influence the scheduler nor can he or she read the offline constructed scheduling tables. Instead, the attacker has to infer the current schedule from observations he or she makes about the system behavior. As we show in Section 3.2, 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 clock remains under exclusive control of the operating system and that it cannot be affected by the attacker.

Even though time-triggered systems eliminate CPU time as shared resource over which information can be leaked and through which other tasks may be influenced, other resources remain through which attackers may gain information and through which they can impact the

[3] Depending on the system configuration, both data and instruction caches may exhibit similar channels or interference possibilities (e.g., instruction cache evictions to maintain inclusiveness in the last-level cache). We shall therefore not further distinguish the type of cache.

timing behavior of other tasks. One prominent example of such a resource is the processor cache, which healthy tasks leave behind in a predictable state but which compromised tasks can put into a state that may not have been anticipated when computing the worst-case execution time of subsequent tasks.

The use of time-triggered systems imposes further limitation on attackers. For example, side channels and covert channels can only be constructed over explicitly or implicitly shared resources, most of which time-triggered systems already multiplex with the table driven schedule in a manner that is agnostic to the behavior of executing tasks. Access controls and partitioning techniques like cache coloring [29] or bank coloring [59] further constrain the attacker. However, each such countermeasure negatively impacts system performance and they are generally not complete.

For example Bechtel et al. [3] 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, both running on different cores and – except for sharing the cache – in isolation. However, usually there are multiple tasks running in a real-time system and the attacker may not hit the victim in all cases without previously aligning its execution to the victim task. If the attacker has inferred the schedule instead, he or she is able to discern the best point in time to attack (e.g. right before the victim task runs). The attacker can remain stealthy until the time of attack has come when the victim is most vulnerable.

In addition to the above, as we show in greater detail in Section 3.2, mitigating attacks may require avoiding tempting optimizations such as bounding the delay a task can impose through the cache by evaluating its execution pattern. Designers may be tempted to implement optimizations for the sake of increasing performance while neglecting security.

In summary, we assume an attacker who has infiltrated non-critical tasks of the system and wants to infer timing information (i.e. the system schedule) in order to mount a directed attack against a critical victim task.

3.2 Vulnerability Analysis

One of the main vulnerabilities of a time-triggered system lies in its *deterministic behavior*. The schedule is the same offline constructed schedule for every hyperperiod. For each point in time, the task executing is known. An attacker who listens to the schedule over a side channel is able to reconstruct the schedule in reasonable time even when the channel has low bandwidth. The schedule comprises only a few bytes of information, thus even with a very low channel bandwidth of, for example, 1 byte per second the schedule is found out in a matter of a few minutes. As we show in Section 7.2.3, an offline schedule of a real-world system can consist of just 52 bytes. Through the aforementioned channel, the attacker would know the schedule after one minute. Therefore, we reason that timing information can be inferred and focus on mitigating directed attacks under this assumption.

Another vulnerability of real-time systems in general is that *worst case execution time (WCET) derivation* does not take malicious behavior into account. WCET estimated through simulation of the expected behavior of the system does not account for malicious behavior. If a task is infiltrated at runtime and, as shown in [3], starts accessing the cache to create maximum interference for the next task execution, the tasks simulated worst case does not account for this malicious behavior if this behavior is not encountered during uncompromised execution. Prior research on abstract interpretation WCET derivation claims the assumption of cold caches is too pessimistic for a real system and shows methods to achieve tighter and less pessimistic WCET bounds [18], [11]. Such optimizations based on assumptions on task behavior increase the severity of this vulnerability. We have to choose WCET estimates in a way such that they also account for malicious behavior and we have to check the impact of performance optimizations on security.

Another countermeasure, typically applied for accidental faults, but found to be effective also against malicious faults aims to limit attacks only to up to f out of n replicas of critical services. This is achieved by tolerating the attack through the remaining $n - f$ replicas operating in consensus (e.g., triple-modular redundant systems [20] tolerate one fault with $n = 3$ replicas). The condition for such replicated systems to work is a synchronchonous invocation with the same (sensor) value such that healthy instances produce the same result in a timely manner, which forms the majority decision to apply. Assuming synchrony (i.e., reliable time), Byzantine fault tolerant state machine replication (BFT-SMR) [2, 28, 40] relaxes the first assumption of synchronous identical invocation at the cost of having to execute an initial agreement phase on a singular value or a vector median point (excluding up to f outliers on both ends) [33].

Given that replicas should operate on the same data to produce (at least approximately) the same results, it is tempting to schedule them as a gang in the same slots on all cores they span. However, as we shall see in more detail in Section 6, this optimization is brittle and may lead to attacks. We shall see in particular that some attacks, like the above cache DOS attack by Bechtel et al. [3], bear the potential to cause common mode timing faults in all replicas simultaneously.

In the next section, we show mitigation strategies for directed attacks which prevent an attacker from exploiting the vulnerability which results from not taking malicious behavior into account.

4 Mitigation Strategies

An attacker's goal is to predict as precisely as possible when a victim task gets scheduled immediately after a compromised task to then mount a directed attack. Our primary mitigation strategy is therefore to impede predictions about the point in time when the victim is executed. While we do not prevent timing inference, i.e. we assume the attacker may gain information about the schedule, 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 section. The first strategy takes an offline constructed time-triggered schedule as input and randomizes the schedule online at job-level while maintaining deadline constraints. The second strategy consists of a set of offline precomputed schedules one of which is randomly chosen at the end of each hyperperiod during runtime. Both strategies are presented in [27] and are implemented on each core of the system.

4.1 Slot-level Online Randomization

This mitigation strategy impedes the ability of an attacker to make predictions by randomizing job execution in a time-triggered system at runtime. Schedules for time-triggered systems are typically constructed offline [9], where real-time constraints are resolved and represented in a scheduling table. If not handled properly, online randomization may violate deadline constraints. Therefore, our approach analyzes the scheduling table 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 task has to finish execution within its execution window. Proper handling and, possibly, modification of execution windows solves precedence constraints. Additionally, if one of the goals of the system is to achieve low jitter, we can reduce the size of execution windows accordingly.

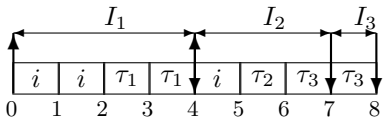
During runtime, we randomize job execution within their respective execution windows. While we confine jobs to their execution windows, they still share the same processor so we also have to guarantee that their execution does not lead to a deadline miss of other jobs. Slot shifting is a scheduling algorithm which introduces the concept of spare capacities to ensure timely execution [12]. We adopt this concept to guarantee task execution within their respective execution windows even though the scheduling decision is randomized.

4.1.1 Background

Slot shifting uses a discrete time model [24], where the time interval which separates two successive events (i.e. the granularity of the system) is called a slot [42]. We analyze 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. The start of a capacity interval I_j, $start(I_j)$, is defined 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:

$$start(I_j) = \max(end(I_{j-1}), est(I_j)), \text{ where } est(I_j) = \min(est(\tau_i)) \ \forall \tau_i \in I_j \tag{2}$$

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. Figure 1 shows an example job set derived from an offline schedule with earliest start times est_i, worst case execution times C_i and deadlines d_i. We derive the presented schedule in Section 4.1.3. In the schedule presented in Figure 1, i denotes the idle task. 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.



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

Figure 1 Job set and capacity intervals derived from offline schedule

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 and ends at the deadline of its assigned set of jobs $\{\tau_1\}$, which is 4. The job assigned to next interval, τ_2, shares the earliest start time of τ_1, but according to Equation 2, 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 together with an exemplary schedule in Figure 1.

The spare capacity $sc(I_j)$ of a capacity interval I_j is equal to the amount of free slots in I_j. $sc(I_j)$ is defined 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), see Equation 3 below.

$$sc(I_j) = |I_j| - \sum_{\tau_i \in I_j} C_i + \min\left(sc(I_{j+1}), 0\right) \tag{3}$$

Spare capacities are calculated starting from the latest capacity interval in the hyperperiod to the earliest. Borrowing occurs in those cases where the current capacity interval provides insufficient slots to accommodate all its jobs, which results in a negative spare capacity (I_3 in Figure 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. Spare capacities are a means to track "free" slots in a capacity interval. We show the resulting offline calculated spare capacities (for time $t = 0$) in Figure 2 of Section 4.1.3, where we present the calculation of spare capacities using Equation 3.

If we have calculated all spare capacities, the first capacity interval has a non-negative spare capacity provided the task set is schedulable. A task set is schedulable when its utilization is equal to or less than one since we consider each core of a partitioned multicore system separately. Positive spare capacities represent the amount of unused resources and leeway [12] 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. 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.
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, but executing the job ahead of time 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 a similar fashion.

The original slot shifting algorithm in [12] and [42] further integrates aperiodic tasks into a time-triggered schedule. In this paper, we only 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 offline algorithm needs to create only one table with execution windows and a second one with capacity intervals and their respective spare capacities. For our online randomizing scheduler, we update the spare capacities at runtime to keep track of scheduling decisions.

4.1.2 Slot-Level Randomization of Jobs

Our first attack mitigation strategy is to randomize job execution at runtime. Therefore, 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. 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 ready job has the same probability of getting selected for a 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 of the current capacity interval. After the job has run during its assigned slot, we update spare capacities as shown in Section 4.1.1.

Combining time-triggered scheduling with our slot-level randomization impedes online 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 [51]. An investigation of leakage countermeasures for such resources is out of the scope of this paper. While our randomization algorithm does not allow for slot-level determinism typical for time-triggered systems, it still allows for execution window determinism [13].

4.1.3 Example

Let us illustrate the proposed scheduling algorithm for our example jobset depicted in Figure 1. First, we have to calculate the initial spare capacities of the capacity intervals. 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 spare capacity of -1. I_2 has an interval length of 3, from which we substract 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 2 shows the resulting 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

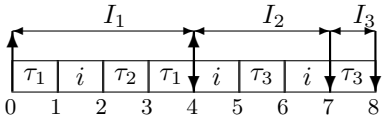


Figure 2 Left: Spare Capacities of I_1, I_2 and I_3 over time, Right: Randomized Schedule

At time $t = 0$, the scheduler sees that the spare capacity of the current interval I_1 is positive and picks τ_1 randomly for the first slot at $t = 0$ from the list of ready jobs τ_1, τ_2, plus 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 is selected 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 2.

4.2 Offline Schedule-Diversification

The second mitigation strategy we investigate in this work precomputes multiple schedules offline and switches between them randomly at hyperperiod boundaries during runtime. Resolving scheduling constraints offline ensures lower runtime overheads, but increases the chance of attackers to guess the schedule and launch directed attacks. For example, repeating the same offline computed schedule several times allows an attacker to infer the schedule, as illustrated in [36], and 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 the attacker is able to recognize different schedules and has enough memory available to store them, the more schedules have been

generated, the harder it is for the attacker to recognize which schedule has been chosen for the current hyperperiod and the less time remains to perform a directed attack. In particular, if the attacker is not able to identify the current schedule in time for his attack, the attacker misses the opportunity to perform a directed attack. Additionally, carefully created execution windows solve deadline and precedence constraints.

We show in Section 7.2.5 that computing and storing all possible, feasible schedules in memory is impractical. However, in non-embedded systems (e.g., SCADA), we foresee the continuing generation of schedules in a non real-time subsystem (e.g., in a sufficiently protected external control station) and an update of the set of schedules downloaded to the real-time device. This way, once a new set of schedules has been produced (possibly by recombining precomputed and stored schedules), the real-time device 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 as much as possible. We present two criteria to select subsets that complicate directed attacks in addition to guaranteeing deadlines and respecting task precedence constraints.

4.2.1 Random Selection

For the sake of low implementation complexity, the subset can be selected randomly. That is, schedules are created randomly and checked to meet all scheduling constraints. The schedules fulfilling this requirement form the set of schedules for the system. Schedule creation is stopped after a certain number of feasible schedules has been constructed. We recommend this method for large subsets, when enough memory is provided 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.

4.2.2 Schedule Entropy

Another criterium for schedule selection is schedule entropy as presented in [58]. This metric makes use of an approximation of the Shannon entropy and can be used to quantify the diversity between schedules. Using an entropy-like function, we can quantify what is the diversity of a set of schedules and then we can compute the set of schedules that maximizes the diversity. This allows us to randomly pick a schedule in this set and give the attacker the least possible amount of information (because from the outside, the task set execution would seem as diverse as possible).

Clearly, deadline and period constraints limit the amount of diversity that can be achieved. In Section 5 we investigate the fundamental limitations to the achievable diversity that are imposed by the task set characteristics.

5 Fundamental limitations to diversity and randomization

In this section, we analyze the schedule of each core separately. For each of them, our final objective is to determine a set $\mathcal{K}$ of k schedules that is as *diverse* as possible, keeping k as small as possible. What we are trying to assess is how effective is the randomization, given that the schedules should respect the time constraints. We are then trying to determine how diverse can a generic set of schedules be, given the constraint that it preserves the deadlines of all the tasks in the task set.

The diversity of the set of selected schedules $\mathcal{K}$ can be measured in different ways. The authors of [58] propose the use of entropy-like functions to measure the diversity of the schedule set, the *slot entropy* and the *upper-approximated entropy*. In the following, we are going to borrow these definitions to evaluate the schedule set diversity. We first define the slot count $C_{j,i,\mathcal{K}}$, as the number of occurrences of a task i in a given scheduling slot j in the set $\mathcal{K}$, and then use that to formally define the slot entropy $H_j(\mathcal{K})$ and the upper-approximated entropy $\tilde{H}(\mathcal{K})$. We denote with $\phi(x)$ the function

$$\phi(x) = \begin{cases} 0 & x \leq 0 \\ -x \cdot \log_2(x) & x > 0 \end{cases}.$$

▶ **Definition 1.** Given a set of k valid schedules $\mathcal{K} = \{s^{(1)}, s^{(2)}, \ldots, s^{(k)}\}$ for the task set $\mathcal{T}$, the j-th time unit, and the i-th task τ_i, we define the slot count $C_{j,i,\mathcal{K}}$ as a function that counts the occurrences of the task i in the given position j in the set $\mathcal{K}$. Using the square brackets as the Iverson brackets — that evaluates to 1 if the proposition inside the bracket is true, and to 0 otherwise — we can then write $C_{j,i,\mathcal{K}}$ as

$$C_{j,i,\mathcal{K}} = \sum_{s^{(q)} \in \mathcal{K}} \left[s_j^{(q)} = i \right] = \sum_{q=1}^{k} \left[s_j^{(q)} = i \right]. \tag{4}$$

Using the slot count, we can now formally write the slot entropy and the upper-approximated entropy according to the definitions given in [58].

▶ **Definition 2.** The slot entropy $H_j(\mathcal{K})$ can be written as a function of the tasks found in slot j, i.e.,

$$H_j(\mathcal{K}) = \sum_{i=0}^{m} \phi\left(\frac{C_{j,i,\mathcal{K}}}{k}\right) = \sum_{i=0}^{m} -\frac{C_{j,i,\mathcal{K}}}{k} \cdot \log_2 \frac{C_{j,i,\mathcal{K}}}{k}. \tag{5}$$

▶ **Definition 3.** The upper-approximated entropy is the sum of all the slot entropies in the hyperperiod, i.e.,

$$\tilde{H}(\mathcal{K}) = \sum_{j=1}^{\ell} H_j(\mathcal{K}) = \sum_{j=1}^{\ell} \sum_{i=0}^{m} -\frac{C_{j,i,\mathcal{K}}}{k} \cdot \log_2 \frac{C_{j,i,\mathcal{K}}}{k}. \tag{6}$$

Now we can formally state the objective of minimizing the information that can be extracted from the system by observing its schedule. Given a task set $\mathcal{T}$ and the set of all valid schedules $\mathcal{S}$, what is the smallest subset of $\mathcal{S}$ that maximizes the upper-approximated entropy?

Mathematically, this problem can be written as the following optimization problem.

▶ **Problem** 4. Given a task set $\mathcal{T}$ and a valid set of schedules $\mathcal{S}$, solve

$$\mathcal{K}^* = \arg\min_{\mathcal{K} \subseteq \mathcal{S}} |\mathcal{K}| \quad \text{s.t. } \tilde{H}(\mathcal{K}) = \max_{\mathcal{L} \subseteq \mathcal{S}} \tilde{H}(\mathcal{L}).$$

Here $\mathcal{L}$ is any generic subset of $\mathcal{S}$, and we search for the set $\mathcal{K}$ with the minimum cardinality that achieves the maximum upper-approximated entropy. This problem consists of two main aspects. First we must determine the maximum upper-approximated entropy; that is we must solve

$$\tilde{H}^* = \max_{\mathcal{L} \subseteq \mathcal{S}} \tilde{H}(\mathcal{L}).$$

Then we must find the smallest subset $\mathcal{K}^* \subseteq \mathcal{S}$ with upper-approximated entropy $\tilde{H}(\mathcal{K}^*) = \tilde{H}^*$.

Problem 4 could be approached by an exhaustive search. If one evaluated the upper-approximated entropy for every element in the power set of $\mathcal{S}$, the optimal solution to Problem 4 could be obtained by choosing the smallest subset that achieves $\tilde{H}^*$. However since the cardinality of $\mathcal{S}$ is typically large, this will be infeasible in practice. Such an approach is also naïve. Afterall it seems improbable that the solution to Problem 4 would have cardinality one, so we could rule those subsets out of our exhaustive search.

One natural question then arises: Are there any fundamental limits on the achievable maximum upper-approximated entropy or the cardinality of $\mathcal{K}$? The remainder of this section is devoted to answering this question. We show that the properties of the task set $\mathcal{T}$ impose fundamental limits both on $\tilde{H}^*$, and on the cardinality of the subsets that can achieve this bound.

The fact that each task in the task set $\mathcal{T}$ must be executed with a certain frequency imposes a fundamental limit on the maximum upper-approximated entropy. The intuitive explanation for this is as follows. It is our objective to select a set of schedules that minimizes the information that an attacker can obtain by observing the execution of any individual task. We therefore want it to appear as if each task was allocated randomly to each given slot. However we cannot necessarily make this allocation appear random with equal probability, because we are required to execute tasks for given time units a certain number times in each hyperperiod and only in the first time units in the period, to meet a given deadline. Therefore the best we can do is make each task appear random with probability specified by its relative frequency in the hyperperiod and not after its deadline is expired. The entropy of the corresponding random variable then specifies an upper bound on the upper-approximated entropy of any set of schedules $\mathcal{K} \subseteq \mathcal{S}$.

▶ **Theorem 5.** *Given any schedule set $\mathcal{K} \subseteq \mathcal{S}$ for a constrained deadline task set,*

$$\tilde{H}(\mathcal{K}) \leq \ell \cdot \sum_{i=0}^{m} \frac{d_i}{t_i} \cdot \phi\left(e_i/d_i\right) =: \tilde{H}^{ub} \tag{7}$$

Proof. The proof hinges on three key observations. The first is that since the function $\phi(x) = -x \cdot \log_2(x)$ is continuous and concave for all $x > 0$, given any set of positive weights $a_j \geq 0$,

$$\sum_{j=1}^{\ell} a_j \phi\left(\frac{C_{j,i,\mathcal{K}}}{k}\right) \leq \left(\sum_{j=1}^{\ell} a_j\right) \phi\left(\frac{\sum_{j=1}^{\ell} a_j C_{j,i,\mathcal{K}}/k}{\sum_{j=1}^{\ell} a_j}\right). \tag{8}$$

The second is that the i-th task can only be active during the first d_i slots of the task period t_i. Therefore since $\phi\left(0\right) = 0$,

$$\phi\left(\frac{C_{j,i,\mathcal{K}}}{k}\right) = I_{j,i}\phi\left(\frac{C_{j,i,\mathcal{K}}}{k}\right), \tag{9}$$

where $I_{j,i} = 1$ if $\mod\ (j-1, t_i) < d_i$, and $I_{j,i} = 0$ otherwise (i.e. $I_{j,i}$ indicates whether the i-th task can be active in slot j). By setting $a_j \equiv I_{j,i}$, together (8)–(9) imply that

$$\sum_{j=1}^{\ell} \phi\left(\frac{C_{j,i,\mathcal{K}}}{k}\right) \leq \left(\sum_{j=1}^{\ell} I_{j,i}\right) \phi\left(\frac{\sum_{j=1}^{\ell} C_{j,i,\mathcal{K}}/k}{\sum_{j=1}^{\ell} I_{j,i}}\right). \tag{10}$$

The final observation is that

$$\sum_{j=1}^{\ell} \frac{C_{j,i,\mathcal{K}}}{k} = \ell\frac{e_i}{t_i}, \quad \sum_{j=1}^{\ell} I_{j,i} = \ell\frac{d_i}{t_i}. \tag{11}$$

These relations follow from the fact that $\mathcal{K}$ contains k valid schedules (i.e. task i is given e_i slots per period t_i, and can only be active in the first d_i slots). Substituting (11) into (10) shows that

$$\sum_{j=1}^{\ell} \phi\left(\frac{C_{j,i,\mathcal{K}}}{k}\right) \leq \left(\ell \frac{d_i}{t_i}\right) \phi\left(\frac{e_i}{d_i}\right).$$

The result follows by summing over all the tasks (c.f. the definition of the upper-approximated entropy). ◀

▶ Remark. Notice that in the case of an implicit deadline task set ($d_i = t_i, \forall i$), the condition of the theorem simplifies to

$$\tilde{H}(\mathcal{K}) \leq \ell \cdot \sum_{i=0}^{m} \phi\left({}^{e_i}/_{t_i}\right). \tag{12}$$

We now present two corollaries that link the upper-approximated entropy to aggregate characteristics of the task set, i.e., the number of tasks and the utilization.

▶ **Corollary 6.** *Given any schedule set $\mathcal{K} \subseteq \mathcal{S}$,*

$$\tilde{H}(\mathcal{K}) \leq -\ell \cdot \log_2\left({}^{1}/_{(1+m)}\right). \tag{13}$$

Proof. Knowing that ${}^{e_i}/_{t_i} > 0$, $\phi({}^{e_i}/_{t_i}) = -{}^{e_i}/_{t_i} \cdot \log_2({}^{e_i}/_{t_i})$ is continuous and concave. This implies that ${}^{1}/_{(m+1)} \sum_{i=0}^{m} \phi({}^{e_i}/_{t_i}) \leq \phi\left({}^{1}/_{(m+1)} \sum_{i=0}^{m} {}^{e_i}/_{t_i}\right)$. This gives us

$$\sum_{i=0}^{m} \phi({}^{e_i}/_{t_i}) \leq (m+1)\, \phi\left({}^{1}/_{(m+1)}\right) = -\frac{m+1}{m+1} \cdot \log_2\left({}^{1}/_{(m+1)}\right).$$

◀

The expression in Equation (13) is not always reachable, depending on the characteristics of the task set. This leads us to consider the task set characteristics. In particular, we can compute a bound that takes into account the utilization U of the task set, leading to the following corollary.

▶ **Corollary 7.** *Given any schedule set $\mathcal{K} \subseteq \mathcal{S}$,*

$$\tilde{H}(\mathcal{K}) \leq \ell \cdot \{-(1-U) \cdot \log_2(1-U) - U \cdot \log_2({}^{U}/_{m})\}. \tag{14}$$

This corollary can be proven by splitting the contribution of the idle task and the regular tasks in the task set. The contribution of the idle task to the upper-approximated entropy is equal to $\ell \cdot \{-(1-U) \cdot \log_2(1-U)\}$. The maximum value for the upper-approximated entropy is reached when the utilizations of the tasks allow them to be evenly distributed. The contribution of each of them is then $-\ell \cdot {}^{U}/_{m} \cdot \log_2({}^{U}/_{m})$. Deriving the expression in Equation (14) allows us also to study when we can be closer to the bound in Equation (13) — and when can we expect to reach the maximum upper-approximated entropy for a set of m tasks, depending on the task characteristics. With respect to the utilization $U = \sum_{i=1}^{m} {}^{e_i}/_{t_i}$, the upper approximated entropy can reach its maximum when the utilization of the system is equal to $U = {}^{m}/_{(1+m)}$.

We will now show that a given schedule set $\mathcal{K} \subseteq \mathcal{S}$ can only achieve an upper-approximated entropy of $\tilde{H}^{ub}$ (from Equation (7)) if all the tasks have implicit deadlines and the cardinality of $\mathcal{K}$ is at least

$$\frac{\ell}{\gcd({}^{e_i}/_{t_i} \cdot \ell)}.$$

This is important, since it shows that if we want to achieve the upper bound on the upper-approximated entropy from Theorem 5, we must enforce $d_i = t_i, \forall i$ and use a schedule set of at least the size given above.

▶ **Theorem 8.** *Given any schedule set $\mathcal{K} \subseteq \mathcal{S}$ for a constrained deadline task set, if there exists a task p such that $d_p < t_p$ then the inequality in Equation (7) is strict. Furthermore, if all the tasks have implicit deadlines ($d_i = t_i, \forall i$) and*

$$|\mathcal{K}| < \frac{\ell}{\gcd\left(e_i/t_i \cdot \ell\right)},$$

then the inequality in Equation (7) is strict.

Proof. The upper-approximated entropy is the sum of the contribution of each slot. Therefore, to achieve the upper bound $\tilde{H}^{ub}$, we should maximize all the summands – i.e., the contribution to the upper-approximated entropy of each slot should be maximized.

It follows from Jensen's inequality and Equation (11) that the inequality in Equation (8) is strict unless

$$C_{j,i,\mathcal{K}}/k = \alpha \tag{15}$$

for all j such that $I_{j,i} = 1$ (reusing the notation of the proof of Theorem 5). The constant α can be found using Equations (10)–(11). In particular, Equation (10) implies that

$$\sum_{j=1}^{\ell} C_{j,i,\mathcal{K}}/k = \alpha \sum_{j=1}^{\ell} I_{j,i} = \alpha \ell \frac{d_i}{t_i} \quad \underset{\text{Equation (11)}}{\Longrightarrow} \quad \alpha = \frac{e_i}{d_i}$$

Since every task is active in the first slot, this implies that the inequality in Equation (7) is strict unless

$$C_{1,i,\mathcal{K}}/k = \frac{e_i}{d_i}. \tag{16}$$

To prove the first part of the theorem, suppose that $d_i < t_i$ for some i. Now assume that Equation (16) holds. This implies that

$$\sum_{i=0}^{m} C_{1,i,\mathcal{K}}/k = \sum_{i=0}^{m} \frac{e_i}{d_i} > 1.$$

However $\sum_{i=0}^{m} C_{1,i,\mathcal{K}} = k$, since every slot in the schedule set is assigned to one task. This is a contradiction.

To prove the second part, now suppose that $d_i = t_i$ for all i. The contribution of each task to the slot entropy should then be equal to the task utilization e_i/t_i – notice that this includes the idle task. To find a lower bound for k, we can then look at a single slot j. In fact, additional slots will only potentially increase the number of schedules needed to achieve higher upper-approximated entropy. We can then formulate the problem of finding $|\mathcal{K}^*| = k^*$ as an optimization problem.

$$\min_{\substack{k \in \mathbb{Z}^+ \\ C_{j,i,\mathcal{K}} \in \mathbb{Z}^+}} \quad k \quad \text{s.t.} \quad C_{j,i,\mathcal{K}} = \frac{e_i}{t_i} \cdot k, \quad \forall i \in \{0, \ldots, m\}. \tag{17}$$

This means that we have a positive integer number of schedules in the set $\mathcal{K}$ that allows $C_{j,i,\mathcal{K}}$ (the number of times task i appears in slot j in set $\mathcal{K}$) to be positive a integer number for each task i. We perform a variable substitution and define $y = \ell/k$. Minimizing k now becomes equivalent to maximizing y. Relaxing temporarily the requirement that k be an integer, the problem in Equation (17) is reformulated as

$$\max_{C_{j,i,\mathcal{K}} \in \mathbb{Z}^+} \quad y \quad \text{s.t.} \quad C_{j,i,\mathcal{K}} = \frac{e_i}{t_i} \cdot \frac{\ell}{y}, \quad \forall i \in \{0, \ldots, m\}. \tag{18}$$

The solution to the problem in Equation (18) is that y should be equal to the greatest common divisor of the utilizations multiplied by the hyperperiod, $y = \gcd(e_i/t_i \cdot \ell)$, which yields to

$$k^* = \frac{\ell}{\gcd\left(e_i/t_i \cdot \ell\right)}. \tag{19}$$

For this to be the solution of the optimization problem in Equation (17), k^* must be an integer number. Because the utilizations of the task set (including the idle task) sum to one, $\sum_{i=0}^{m} e_i/t_i = 1$, the constraint for the idle task in the optimization problem of Equation (18) can also be written as

$$C_{j,0,\mathcal{K}} = \left(1 - \sum_{i=1}^{m} \frac{e_i}{t_i}\right) \cdot \frac{\ell}{y} = \left(\frac{\ell}{y} - \frac{\ell}{y} \cdot \sum_{i=1}^{m} \frac{e_i}{t_i}\right).$$

The solution of problem (18) ensures that $\frac{\ell}{y} \cdot \sum_{i=1}^{m} e_i/t_i \in \mathbb{Z}^+$ due to the m constraints for the (non idle tasks in the) task set. The value of ℓ/y (equal to k^*) must then be a positive integer number. This implies that the solution of the problem in Equation (18) is also a solution of the problem in Equation (17) and $k^* \in \mathbb{Z}^+$. ◀

▶ **Corollary 9.** *A simple modification of the proof of Theorem 8 shows that for any $\mathcal{K} \subseteq \mathcal{S}$, if*

$$|\mathcal{K}| \mod \frac{\ell}{\gcd(e_i/t_i \cdot \ell)} \neq 0,$$

then $\tilde{H}(\mathcal{K}) < \tilde{H}^{ub}$. This means that the cardinality of $\mathcal{K}$ must be a multiple of $\frac{\ell}{\gcd(e_i/t_i \cdot \ell)}$ in order to achieve the bound in Theorem 8.

6 Extension to replicated systems on multicores

So far, we have assumed that accomplices of attackers are limited to the tasks of non-critical subsystems. Through the use of replication, we can consider stronger adversaries, capable of compromising also up to f replicas of an n-fold replicated critical subsystem. This leads to the following additional attack vectors:

AV1: A compromised task or replica attacks the task executing next on the same core.
AV2: A compromised task or replica attacks replicas executed in parallel on another core.
AV3: A combination of both attack vectors AV1 and AV2, i.e., attacks are mounted from one core shortly before a replica is run on another core.

The challenge with AV2 and AV3 lies in ensuring that at most f replicas can be affected in any cycle required to rejuvenate all n replicas, returning them to a healthy state [45, 46]. In particular in real-time systems, where results must be timely, systematic delays of more than f replicas (minus those already compromised) may turn out fatal, as this allows compromised but timely replicas and consequently the attacker to gain control. Extending offline and online schedule-diversification allows us to protect against the above threat vectors.

6.1 Offline Schedule-Diversification

Offline schedulers posess the advantage to solve complex constraints before runtime. In addition to the deadline constraints already discussed in Section 4.1.1 (and possibly further constrains to accomodate for precedence or jitter), we shall impose as constraint that at the end of each slot, no two cores 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 cores RTOS stores all schedules. The RTOS instances on these cores switch at the end of the hyperperiod to another schedule in an offline defined, deterministic way, such that each core executes a different schedule. This still provides deterministic execution to the attacker, however 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 enough, the possibility of retrieving information using the schedule is low. In addition, if the upper-approximated entropy bound given by Equation (7) is reached for the set of schedules that is used for schedule diversification, it is possible to state that there is no additional diversity that can be added by varying the schedule.

Cores may 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.

6.2 Online Slot-Level Randomization

Online slot-level randomization, as discussed in Section 4.1.2, prevents predictions about the points in time when replicas are executed in parallel or on other cores and thus mitigates all attack vectors, AV1 to AV3. The attacker is unable to predict system behavior and cannot rely on deterministic assumptions to coordinate his or her attack. This strategy prevents the stealthy attack we consider for all three attack vectors.

6.3 Combining Offline and Online Strategies

In this approach, we combine both strategies presented in Section 6.1 and Section 6.2. First we create non-overlapping execution windows of replicas offline, i.e., replica execution windows are restricted not to overlap each other to fulfill this constraint. As consequence, the 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 slot-level online 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 6.2, 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 Experiments

We evaluate our mitigation strategies and insights on schedule entropy with experiments. First, we present a search algorithm that generates a set of schedules for each synthetic task set under the constraint that it maximizes the task sets upper-approximated entropy. We investigate the limits on randomization and entropy through the achievable entropy and its variance. Second, we employ the ROSACE case study [38] to evaluate our attack mitigation strategies. The ROSACE case study provides task set parameters for a safety-critical real-time system of an aircraft called the longitudinal flight controller. We provide an analysis of runtime overhead and memory cost.

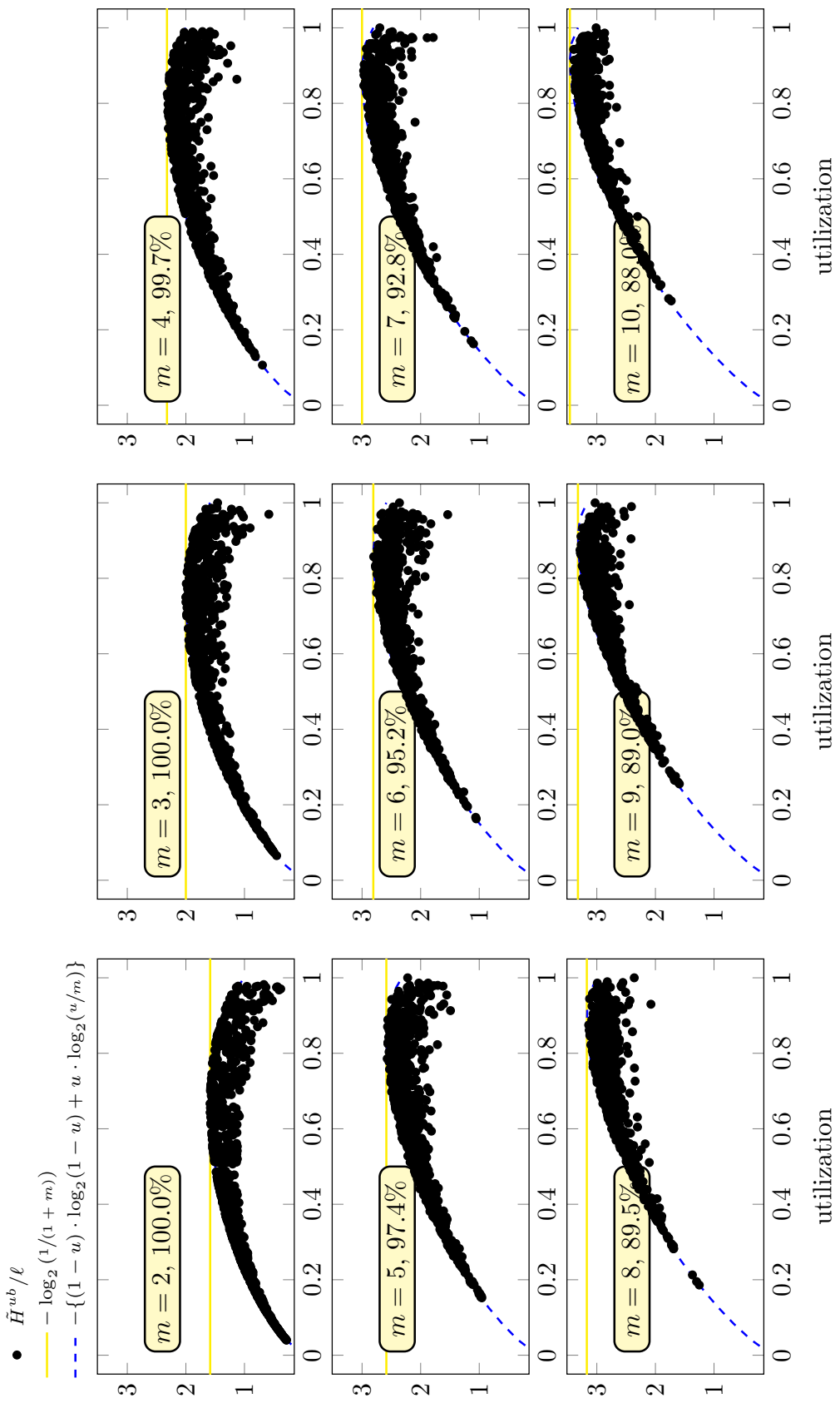


Figure 3 Average slot entropy of random sets composed of m tasks with $m \in \{2, \ldots, 10\}$ and maximum hyperperiod $\ell = 100$.

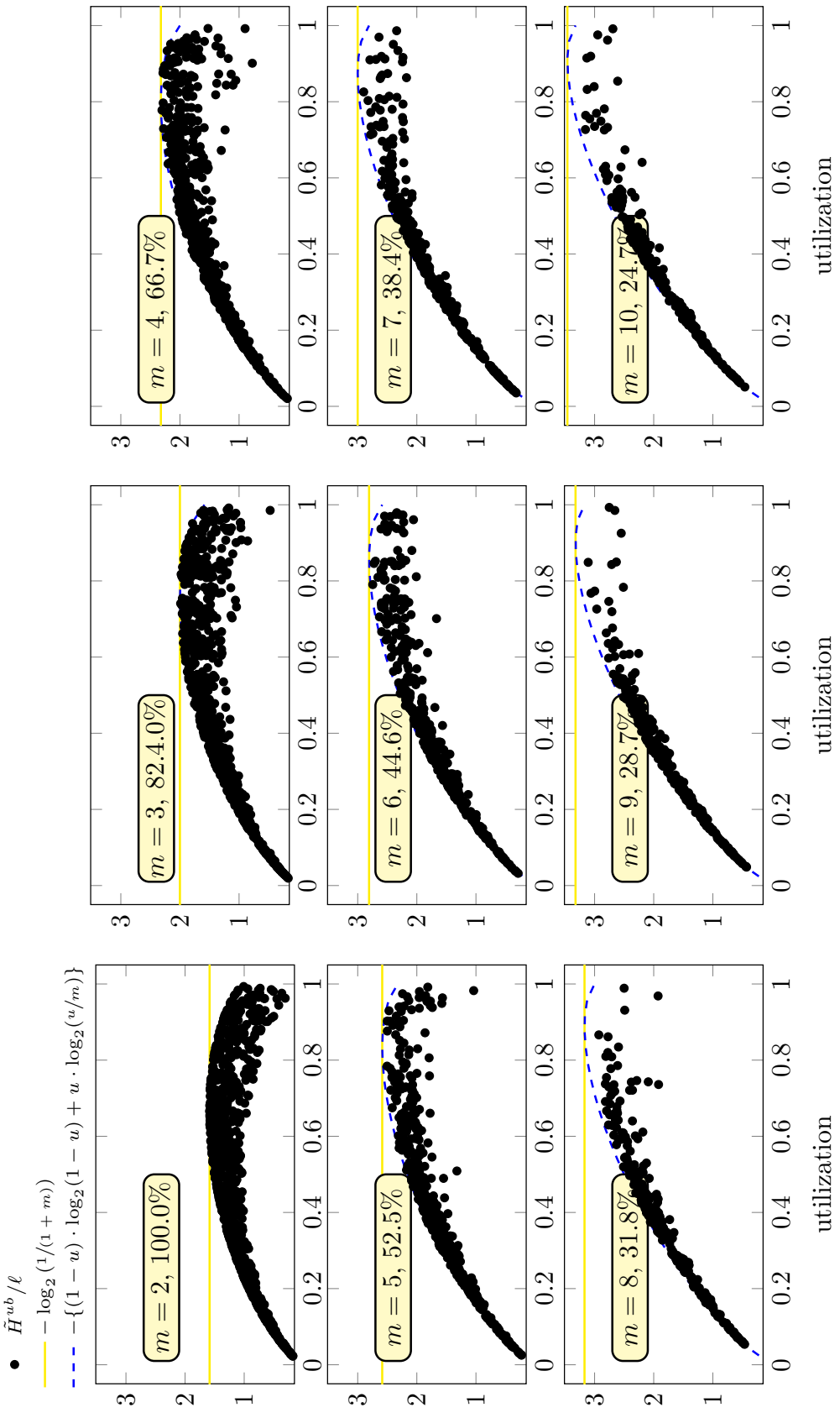


Figure 4 Average slot entropy of random sets composed of m tasks with $m \in \{2, \ldots, 10\}$ and maximum hyperperiod $\ell = 500$.

7.1 Synthetic Task Sets

In order to investigate the limits on schedule randomization and schedule entropy presented in Section 5, we implemented a search algorithm based on constraint optimization that generates a set of schedules maximizing the upper-approximated entropy for a given task set.[4]

We looked at each core separately and tested the algorithm with synthetic task sets, composed of n tasks with $n \in \{2, \ldots, 10\}$ and maximum hyperperiod $\ell \in \{100, 200, 300, 400, 500\}$. For each of the possible combinations, we generated 1000 task sets using an extension of the UUniFast algorithm [4], that allows us to control the maximum hyperperiod. We randomized the periods t_i of each task, and imposed an implicit deadline constraint $t_i = d_i, \forall i \in \{1, \ldots, n\}$. We computed the upper-approximated entropy $\tilde{H}^{ub}$, for the task set according to Theorem 5. We then computed the average contribution of each slot $\tilde{H}^{ub}/\ell$, to be able to compare task sets with different hyperperiods. Finally, we ran our algorithm to generate the schedule set $\mathcal{K}$, with the lowest cardinality k^* given by Theorem 8.

In principle, the algorithm could run for a long time to find the maximum. There is also the possibility that the maximum is not reachable with any schedule set that satisfies all the constraints (in our experimental campaign, we did not encouter such a set, but our theoretical results do not exclude this possibility). Therefore, we limited the duration and allowed a budget of 60 minutes to compute the schedule set.

We define the algorithm *accuracy* as the percentage of task sets for which the algorithm found an optimal schedule set. In Figures 3 and 4, we show the results for the cases with $\ell = 100$ and $\ell = 500$, respectively. In particular, we show the average contribution of each slot to the upper-approximated entropy $\tilde{H}^{ub}/\ell$. In each plot, we write the number of tasks m and the algorithm accuracy.

The dots in the Figures 3 and 4 show the tight bounds and the achievable entropy, while the dashed line represents the (non-tight) bound discussed in Corollary 7, Equation (14). As can be seen, for some of the task sets, the constraints imposed by the execution times and the periods allow the upper-approximated entropy to reach this bound, but in other cases the bound presented in Theorem 5 is tighter. One can also notice that the variance of the achieved upper-approximated entropy increases when the utilization increases. This is because a higher utilization introduces tighter constraints on the achievable entropy.

Another important result is the fact that the upper-approximated entropy reaches the maximum bound discussed in Corollary 7 for $m/(m+1)$. The implication of this is that from an entropy perspective the optimal system utilization is $m/(m+1)$. This is very important for determining critical system parameters like the task set utilization for security-aware embedded and real-time systems.

Finally, the yellow solid lines represent the (non-tight) bound introduced by Corollary 6. In most cases, this bound is unreachable, due to the constraints introduced by the tasks characteristics.

7.2 Real-world case study (ROSACE)

We evaluated our two directed attack mitigation strategies presented in Section 4 using the ROSACE case study [38]. ROSACE is a practical, real-world example of a real-time system in a safety-critical avionics environment. Pagetti et al. [38] 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

[4] The code for the algorithm is available at `https://gitlab.control.lth.se/NilsVreman/rand-sched`, together with the full set of results.

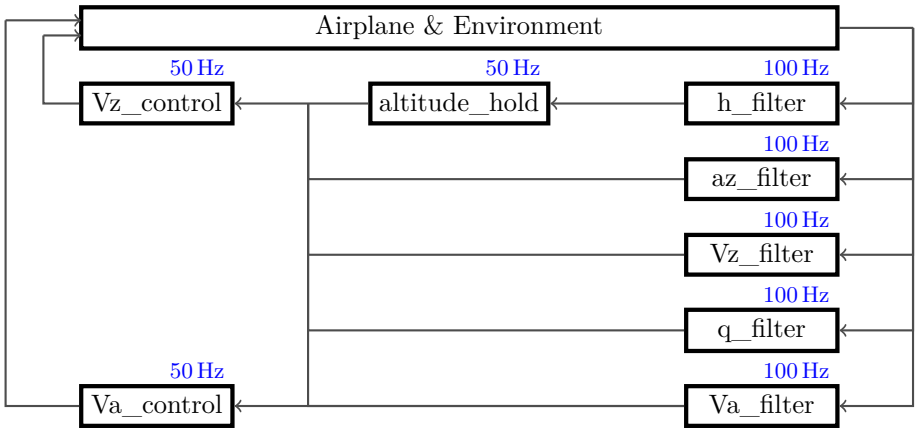


Figure 5 Longitudinal flight controller design

Table 1 Flight controller task set [38]

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

Table 2 Execution windows in terms of slots

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

$Va\_control$ 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 5.

According to Pagetti et al. [38], 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 100μs, thus the worst case execution times for the tasks shown are the same as they presumably finished execution in that granule. Table 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.

We construct the execution windows of all tasks from the task set in Table 1. Schorr [42] 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. Table 2 shows the resulting execution windows.

7.2.1 Runtime Overhead for Slot-Level Randomization

Our slot-level randomization algorithm is based on Schorr's [42] 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. Schorr provided minimum and maximum runtimes of all parts of the slot shifting algorithm for single core execution. Using the timing measurements of [42], shown in Table 3, we approximate the runtime overhead of slot-level randomization, when executed on the same processor.

Table 3 Minimum and maximum runtime overhead per slot for single core execution in ns [42]

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

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 linearly with the number of intervals due to cascaded borrowing in the worst case. However, according to the slot shifting algorithm as explained in Section 4.1.1, only 2 intervals are created for the presented 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 an implementation of slot-level randomization in the same 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 as:

$$t_{ov,rand,max} = rand_{max} + up_{sc,max} + up_{ready,max} + sel_{max} + ISR_{max} \tag{20}$$

Accordingly, the minimum runtime overhead results in:

$$t_{ov,rand,min} = rand_{min} + up_{sc,min} + up_{ready,min} + sel_{min} + ISR_{min} \tag{21}$$

Using the measurements from Table 3 for equation 20 and assuming $rand_{max} = 5,000ns$, the maximum runtime overhead results in $t_{ov,rand,max} = 29,730ns$, which is around 3 percent of the assigned slot size of 1ms in [42]. Keeping in mind that ROSACE uses 6 times faster cores than [42] 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 $t_{ov,rand,max} = 6,000ns$.

Under the assumption that $rand_{min} = 2,000ns$, the minimum runtime overhead results in $t_{ov,rand,min} = 12,565ns$, which is around 1.3 percent of the slot size in [42]. Dividing these values by 5 as explained earlier, we approximate the minimum runtime overhead for ROSACE to be $t_{ov,rand,min} = 2,500ns$.

7.2.2 Runtime Overhead for Offline Precomputed Schedules

The runtime overhead for offline precomputed schedules is lower than that of scheduling algorithms which have to take more complex decisions online, which we also prove in this section. Again we can make use of the overhead measurements done in [42], which we show in Table 4.

Table 4 Minimum and maximum runtime overhead for single core execution in ns [42]

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

At runtime, the scheduler performs a table lookup to select the next job after each slot. In constrast 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 precomputed schedule randomly. We calculate best and worst case runtime overhead for selecting a precomputed schedule in MPARM as shown below.

$$t_{ov,prec,max} = rand_{max} + sel_{max} + ISR_{max} = 10470ns \tag{22}$$

$$t_{ov,prec,min} = rand_{min} + sel_{min} + ISR_{min} = 6410ns \tag{23}$$

Using the same estimation on the execution time of the randomization function for the ROSACE case study as 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 for both ROSACE and on the ARM simulator MPARM.

7.2.3 Memory Cost for Offline Precomputed Schedules

Each precomputed schedule needs to be stored in memory. For ROSACE, we can build an offline schedule in the same way as shown in Table 2. Each task has its own task ID, an entry for the start and end of the execution of its instance, and a fourth entry for its worst case execution time. The difference between start and end time must be equal to its worst case execution time and the execution windows for different jobs must not overlap. Table 5 shows an example for a precomputed time-triggered schedule.

Table 5 Exemplary precomputed time-triggered schedule for ROSACE

ID	0	1	2	3	4	0	1	2	3	4	5	6	7
Start	1	8	22	33	35	51	58	66	67	71	80	88	94
End	2	9	23	34	36	52	59	67	68	72	81	89	95
WCET	1	1	1	1	1	1	1	1	1	1	1	1	1

Assuming each entry has the size of 1 byte, a single schedule with this information needs $13 * 4 = 52$ bytes of memory. Theorem 8 and Corollary 9 state that we need only k^* schedules to achieve the maximum diversity. In the case of ROSACE, this number is 100. Thus, we are able to

calculate the total memory cost to store all these schedules, which amounts to $100 * 52 = 5200$ bytes. This example also shows that the memory cost scales linearly with the number of tasks involved. Each instance of a periodic task, i.e. each job adds 4 entries to this table.

7.2.4 Upper-approximated Entropy Analysis

Here we derive the actual numbers for the ROSACE case study, determining the limitations of the achievable upper-approximated entropy. We first look at the specifics of the given task set, and all the individual task utilizations, using Theorem 5. Then we look at a generic task set with the same utilization as the ROSACE utilization and apply the results of Corollary 7. Here, we see how ROSACE compares with other task sets with the same utilization (unsurprisingly, ROSACE does not have the "perfect" workload distribution to achieve the highest entropy). Notice that our solution is optimal in the space of solutions that satisfy the ROSACE schedulability.

Using a slot length of $200\,\mu$s, we can compute the hyperperiod $\ell = 100$ slots and the minimum number of schedules k^* needed to achieve the maximum value of the upper-approximated entropy according to Equation (18), which amounts to 100 schedules. We use the tool presented in Section 7.1 to generate a set of 100 schedules that achieves the maximum upper-approximated entropy. The generated set reaches a value of $\tilde{H}^* = 93.8495$, which is the optimal value that can be reached for the characteristics of the given task set. This means that each of the schedule slots contributes with a diversity of 0.9385. This number is, again, optimal with respect to the utilizations and periods of the task set of this specific case study.

We now turn to the question how good the given scenario performs in terms of diversity compared to what is possible for task sets of the same utilization. For that, we can look at Corollary 7. The maximum upper-approximated entropy contribution per slot is 0.9474, given by Equation (14). This means that the distribution of the tasks that we find in our scenario is very close to the optimum that can be achieved at the given utilization level.

7.2.5 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 will analyze our mitigation methods with respect to these constraints.

Slot-level randomization proves to be practical, as the approximated overhead in Section 7.2.1 shows. In the worst case, slot-level randomization uses less than 3 percent of the slot size for scheduling. Precomputing offline schedules can further reduce this overhead to roughly 1 percent of the slot size, but physical memory capacity limits the number of offline precomputed schedules that can be stored in a system. In general, it is possible to offload scheduling tables to secondary storage by accepting an increase of scheduling overhead while loading the selected scheduling table from this memory. However, we know from Theorem 8 and Corollary 9 that we need only k^* schedules to achieve the maximum diversity and we are able to calculate this number. In the case of ROSACE, we need 100 schedules to achieve maximum diversity, which amounts to 5200 bytes of required memory storage (see Section 7.2.3 for details).

As we mentioned in Section 3.2, an attacker might identify a small number of schedules after several minutes or a few hours even for side channels with low bandwith. The attacker might even be able to derive a minimal set of schedules that achieves maximum diversity, as scheduling parameters might be known, derived from system observation or reverse engineered. However, this set is not unique, i.e. different sets of schedules are able to achieve maximum diversity. 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 is used. Updating the stored scheduling tables partially mitigates 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.

In order to show how many possible schedules slot-level randomization covers, we calculate the total number of possible feasible schedules for the task set presented in Table 2. 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 substracted from the execution window sizes of the tasks with a period of 100 slots.

$$\left[\binom{50}{1}\binom{49}{1}\binom{48}{1}\binom{47}{1}\binom{46}{1}\right]^2 \times \binom{90}{1}\binom{89}{1}\binom{88}{1} = 4.56 \times 10^{22} \tag{24}$$

4.56×10^{22} schedules with 52 bytes require 2^{81} bytes of storage, so 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.

8 Related Work

Security is a major concern for real-time and control systems [15, 16, 48–50, 57]. Modern embedded systems are vulnerable to many different security threats [23, 39], one of them being side-channel attacks [47]. Side-channel attacks are based on attackers gathering knowledge about a system, and exploiting this knowledge to influence its behavior [1, 34, 41, 44, 47]. For example, recently, a team of researchers showed that it is possible to retrieve the engine speed from the frequency of execution of its control task [31]. In general an attacker knowing the schedule of an embedded control system can infer that the controller is sending a control signal to a plant periodically in predictable time slots. They can then use this knowledge to jam the network only when the control signal is being transmitted. Reducing the need for the attacker to be active also reduces the possibility of detecting the ongoing attack.

Several security solutions exist which prevent information leakage in real-time systems. For example, Völp et al. show in [52] how to 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. Naturally, time-triggered systems do not require this modification since no two tasks execute in the same time window on the same processor. In [36], 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 [52] and [36], we consider

time-triggered systems which have no concept of task priority. Additionally, we do not focus on preventing timing channels or information leakage. In fact, we assume timing information, in particular task set parameters, may be inferred.

One of the logical countermeasures against this type of attacks is to impede the information gathering phase. In particular, an attacker who observes the execution of the real-time system should not be able to get timing information beneficial for an attack. However, classical scheduling algorithms are designed exactly for predictability and repeatability. Schedules (for periodic task sets) usually repeat after a predetermined amount of time. This is precisely what gives an attacker the ability to observe the system and infer knowledge. An observer collecting information for long enough can then infer the execution pattern and rely on the real-time systems predictability for a directed attack. Schedule randomization was proposed [58] to defend real-time systems against side-channel attacks. During the execution of the system, as long as deadline constraints are not violated, the next task is picked randomly from the ready queue. The schedule is either generated online [27, 58], or selected from a set of pre-generated schedules [26, 27]. For embedded systems, the overhead of online generation can be avoided if it is possible to compute and store a schedule set with acceptable diversity [26].

Nasri et al. [37] analyze the conditions for successful time-domain attacks, concluding the difficulty of mounting such attacks in event-triggered systems. While it is true that some of these attacks can be difficult to mount for a generic system, the predictable schedule of time-triggered systems makes them more vulnerable. The attacks are simpler to carry on and can be more disruptive.

Two examples for state-of-the-art research deal with security for time-triggered communication. In [43], Skopik et al. introduce a security architecture for time-triggered communication which adds device authentication, secure clock synchronization and application level security. Wasicek et al. [54] 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 infering timing information, but instead focus on covert or side channels and the implication of attackers learning timing information to coordinate their attacks.

9 Conclusion

In this paper we analyzed vulnerabilities of time-triggered systems with respect to timing-inference based directed attacks, presented two mitigation strategies, and analyzed the randomness of schedules. The deterministic behaviour of time-triggered systems allows attackers to infer timing information over side channels and precisely target victim tasks. Worst case execution time assumptions, on which schedules are based, do not take malicious behaviour into account. As the schedule of a time-triggered system comprises only a few bytes, it can be inferred by an attacker over side-channels. In order to prevent attackers from predictions about the point in time when a certain task is executed, we presented two mitigation strategies for directed attacks. First, we introduced slot-level randomization, which impedes predictions about the schedule by selecting the next job at random. We employ concepts of slot shifting to allow randomization of a time-triggered schedule without violating timing constraints. Secondly, we proposed online selection of offline precomputed schedules for mitigation of directed attacks. At runtime, a schedule from a precomputed set of schedules is randomly selected at the end of each hyperperiod. We showed how to compute the minimum number of schedules needed to achieve maximum schedule diversity and devised an algorithm to find these schedules. First, we tested our algorithm with synthetic task sets and presented results regarding achievable entropy respecting varying hyperperiods and

utilization levels. Then, we evaluated our mitigation strategies with respect to overhead and memory cost with a practical, real-world case study of a safety-critical flight controller. Slot-level randomization has a runtime overhead of around 3 percent of the slot size in the worst case, which makes it suitable for practical use. Scheduling precomputed schedules reduces the worst case runtime overhead to around 1 percent of the slot size, but is more costly in terms of memory. A single schedule for the case study has a size of 52 bytes, but the total number of feasible schedules lies in the magnitude of 10^{22}. Out of this large amount of schedules, only 100 are needed to achieve the optimal upper-approximated entropy. Thus, both mitigation strategies proved to be practical. Attackers could still try to launch undirected attacks, but they will be easier to detect this way.

References

1 Dakshi Agrawal, Bruce Archambeault, Josyula Rao, and Pankaj Rohatgi. 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.

2 Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. Shifting Gears: Changing Algorithms on the Fly to Expedite Byzantine Agreement. *Inf. Comput.*, 97(2):205–233, 1992. doi:10.1016/0890-5401(92)90035-E.

3 Michael G. Bechtel and Heechul 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.

4 Enrico Bini and Giorgio Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2), 2005. doi:10.1007/s11241-005-0507-9.

5 Peter K. Boucher, Raymond K. Clark, Ira B. Greenberg, E. Douglas Jensen, and Douglas M. Wells. *Toward a Multilevel-Secure, Best-Effort Real-Time Scheduler*, pages 49–68. Springer Vienna, Vienna, 1995. doi:10.1007/978-3-7091-9396-9_8.

6 Luis Brandao and Alysson Bessani. On the Reliability and Availability of Systems Tolerant to Stealth Intrusion. In *5th Latin-American Symposium on Dependable Computing (LADC'11)*, Brazil, April 2011. doi:10.1109/LADC.2011.27.

7 Luis Brandao and Alysson Bessani. On the Reliability and Availability of Replicated and Rejuvenating Systems under Stealth Attacks and Intrusions. *Journal of the Brazilian Computer Society*, 18:61–80, March 2012. doi:10.1007/s13173-012-0062-x.

8 Xi Chen, Juejing Feng, Martin Hiller, and Vera Lauer. 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.

9 Silviu S. Craciunas and Ramon Serna 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, pages 45:45–45:54, New York, NY, USA, 2014. ACM. doi:10.1145/2659787.2659812.

10 Joanne Bechta Dugan and Randy Van Buren. Reliability evaluation of fly-by-wire computer systems. *Journal of Systems and Software*, 25(1):109–120, 1994. doi:10.1016/0164-1212(94)90061-2.

11 Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2):131–181, November 1999. doi:10.1023/A:1008186323068.

12 G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 152–161, December 1995. doi:10.1109/REAL.1995.495205.

13 Gerhard Fohler. *Advances in Real-Time Systems, Chapter Predictably Flexible Real-time Scheduling*. SPRINGER, 2012.

14 Alain Girault, Hamoudi Kalla, and Yves Sorel. An active replication scheme that tolerates failures in distributed embedded real-time systems. In *Design Methods and Applications for Distributed Embedded Systems*, pages 83–92. Springer, 2004. doi:10.1007/1-4020-8149-9_9.

15 Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B. Bobba. A design-space exploration for allocating security tasks in multicore real-time systems. In *Design, Automation & Test in Europe*, DATE, 2018. doi:10.23919/DATE.2018.8342007.

16 J.M. Hendrickx, K.H. Johansson, R.M. Jungers, H. Sandberg, and K.C. Sou. Efficient computations of a security index for false data attacks in power networks. *IEEE TAC*, 59(12):3194–3208, 2014. doi:10.1109/TAC.2014.2351625.

17 W. M. Hu. Lattice scheduling and covert channels. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 52–61, May 1992. doi:10.1109/RISP.1992.213271.

18 B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, April 2011. doi:10.1109/RTAS.2011.27.

19 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, 2010.

20 Rolf Isermann, Ralf Schwarz, and Stefan Stolzl. Fault-tolerant drive-by-wire systems. *IEEE Control Systems*, 22(5):64–81, 2002. doi:10.1109/MCS.2002.1035218.

21 Road vehicles – Functional safety, 2011.

22 P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019*

IEEE Symposium on Security and Privacy (SP), pages 1–19, 2019. doi:10.1109/SP.2019.00002.

23 Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. meltdownattack.com, 2018. URL: https://spectreattack.com/spectre.pdf.

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

25 Kristin Krüger, Gerhard Fohler, Marcus Völp, and Paulo Esteves-Verissimo. 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*, August 2018. doi:10.1109/RTCSA.2018.00036.

26 Kristin Krüger, Marcus Völp, and Gerhard Fohler. 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.

27 Kristin Kruger, Marucs Völp, and Gerhard Fohler. 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.

28 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.

29 J. Liedtke, H. Hartig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 213–224, June 1997. doi:10.1109/RTTAS.1997.601360.

30 Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, 2018. arXiv:1801.01207.

31 Songran Liu, Nan Guan, Dong Ji, Weichen Liu, Xue Liu, and Wang Yi. Leaking your engine speed by spectrum analysis of real-time scheduling sequences. *Journal of Systems Architecture*, 2019. doi:10.1016/j.sysarc.2019.01.004.

32 Sixing Lu, Minjun Seo, and Roman Lysecky. 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.

33 Keith Marzullo. Tolerating failures of continuous-valued sensors. *ACM Transactions on Computer Systems*, 8(4):284–304, November 1990. doi:10.1145/128733.128735.

34 Edgar Mateos and Catherine Gebotys. A new correlation frequency analysis of the side channel. In *Workshop on Embedded Systems Security*, 2010. doi:10.1145/1873548.1873552.

35 Robert Mitchell and Ing-Ray Chen. A survey of intrusion detection techniques for cyber-physical systems. *ACM Computing Surveys (CSUR)*, 46(4), 2014. doi:10.1145/2542049.

36 Sibin Mohan, Man-Ki Yoon, Rodolfo Pellizzoni, and Rakesh B Bobba. Integrating security constraints into fixed priority real-time schedulers. *Real-Time Systems*, pages 1–31, 2016. doi:10.1007/s11241-016-9252-5.

37 Mitra Nasri, Thidapat Chantem, Gedare Bloom, and Ryan M. Gerdes. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In Björn B. Brandenburg, editor, *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 103–116. IEEE, 2019. doi:10.1109/RTAS.2019.00017.

38 C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. 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)*, pages 309–318, April 2014. doi:10.1109/RTAS.2014.6926012.

39 Dorottya Papp, Zhendong Ma, and Levente Buttyan. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *13th Annual Conference on Privacy, Security and Trust*, PST, 2015. doi:10.1109/PST.2015.7232966.

40 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.

41 Thomas Popp, Stefan Mangard, and Elisabeth Oswald. Power analysis attacks and countermeasures. *IEEE Design & test of Computers*, 24(6), 2007. doi:10.1109/MDT.2007.200.

42 Stefan Schorr. *Adaptive Real-Time Scheduling and Resource Management on Multicore Architectures*. PhD thesis, Technical University of Kaiserslautern, March 2015. URL: https://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/4008.

43 Florian Skopik, Albert Treytl, Arjan Geven, Bernd Hirschler, Thomas Bleier, Andreas Eckel, Christian El-Salloum, and Armin Wasicek. *Towards Secure Time-Triggered Systems*, pages 365–372. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-33675-1_33.

44 Joon Son and Jim Alves-Foss. Covert timing channel capacity of rate monotonic real-time scheduling algorithm in MLS systems. In *Communication, Network, and Information Security*, 2006. doi:10.1109/IAW.2006.1652117.

45 P. Sousa, N. F. Neves, and P. Verissimo. Proactive resilience through architectural hybridization. In *ACM Symposium on Applied Computing*, pages 686–690, 2006. doi:10.1145/1141277.1141435.

46 Paulo Sousa, Alysson Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery. *IEEE Transactions on Parallel and Distributed Systems, vol. 21, no. 4, pp. 452-465, Apr. 2010.*, 2010. URL: http://www.navigators.di.fc.ul.pt/archive/papers/ieeetpds-prrw-final-version.pdf.

47 Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: A case study for

mobile devices. *IEEE Communications Surveys and Tutorials*, 20(1), 2018. doi:10.1109/COMST.2017.2779824.

48 A. Teixeira, I. Shames, H. Sandberg, and K.H. Johansson. Revealing stealthy attacks in control systems. In *Allerton Conference on Communication, Control, and Computing*, pages 1806–1813, 2012. doi:10.1109/Allerton.2012.6483441.

49 A. Teixeira, I. Shames, H. Sandberg, and K.H. Johansson. Distributed fault detection and isolation resilient to network model uncertainties. *IEEE Transactions on Cybernetics*, 44(11):2024–2037, November 2014. doi:10.1109/TCYB.2014.2350335.

50 Mankuan Vai, Roger Khazan, Daniil Utin, Sean O'Melia, David Whelihan, and Benjamin Nahill. Secure embedded systems. Technical report, MIT Lincoln Laboratory Lexington United States, 2016.

51 M. Völp, B. Engel, C. J. Hamann, and H. Härtig. On confidentiality-preserving real-time locking protocols. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013. doi:10.1109/RTAS.2013.6531088.

52 Marcus Völp, Claude-Joachim Hamann, and Hermann Härtig. Avoiding timing channels in fixed-priority schedulers. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '08, pages 44–55, New York, NY, USA, 2008. ACM. doi:10.1145/1368310.1368320.

53 Nils Vreman, Richard Pates, Kristin Krüger, Gerhard Fohler, and Martina Maggio. Minimizing side-channel attack vulnerability via schedule randomization. In *58th IEEE Conference on Decision and Control (CDC)*, December 2019. doi:10.1109/CDC40024.2019.9030144.

54 A. Wasicek, C. El-Salloum, and H. Kopetz. 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*, pages 31–39, March 2011. doi:10.1109/ISORC.2011.14.

55 Armin Rudolf Wasicek. *Security in Time-Triggered Systems*. PhD thesis, Technische Universität Wien, 2011.

56 C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to Integrated Modular Avionics. In *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, pages 2.A.1–1–2.A.1–10, October 2007. doi:10.1109/DASC.2007.4391842.

57 Steve H. Weingart. Physical security devices for computer subsystems: A survey of attacks and defenses. In *Cryptographic Hardware and Embedded Systems*, CHES, 2000. doi:10.1007/3-540-44499-8_24.

58 Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. 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.

59 H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. 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)*, pages 155–166, April 2014. doi:10.1109/RTAS.2014.6925999.

60 Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. Time-based intrusion detection in cyber-physical systems. In *1st ACM/IEEE International Conference on Cyber-Physical Systems*, 2010. doi:10.1145/1795194.1795210.