Vulnerability Analysis and Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Systems

Kristin Krüger
Technische Universität Kaiserslautern
Kaiserslautern, Deutschland
krueger@eit.uni-kl.de
https://orcid.org/0000-0002-3201-5528

Marcus Völp[1]
SnT - Université du Luxembourg
Esch-sur-Alzette, Luxembourg
marcus.voelp@uni.lu
https://orcid.org/0000-0002-8020-4446

Gerhard Fohler
Technische Universität Kaiserslautern
Kaiserslautern, Deutschland
fohler@eit.uni-kl.de
https://orcid.org/0000-0001-6162-2653

Abstract

Much effort has been put into improving the predictability of real-time systems, especially in safety-critical environments, which provides designers with a rich set of methods and tools to attest safety in situations with no or a limited number of accidental faults. However, with increasing connectivity of real-time systems and a wide availability of increasingly sophisticated exploits, security and, in particular, the consequences of predictability on security become concerns of equal importance. Time-triggered scheduling with offline constructed tables provides determinism and simplifies timing inference, however, at the same time, time-triggered scheduling creates vulnerabilities by allowing attackers to target their attacks to specific, deterministically scheduled and possibly safety-critical tasks. In this paper, we analyze the severity of these vulnerabilities by assuming successful compromise of a subset of the tasks running in a real-time system and by investigating the attack potential that attackers gain from them. Moreover, we discuss two ways to mitigate direct attacks: slot-level online randomization of schedules, and offline schedule-diversification. We evaluate these mitigation strategies with a real-world case study to show their practicability for mitigating not only accidentally malicious behavior, but also malicious behavior triggered by attackers on purpose.

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

Keywords and phrases real-time systems, time-triggered systems, security, vulnerability

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.22

Acknowledgements We want to thank the reviewers for their helpful comments which greatly improved this paper. We are also gratefully indebted to Ali Syed, Florian Heilmann and Rodrigo

[1] supported by Fonds National de la Recherche Luxembourg (FNR) through PEARL grant FNR/P14/8149128.

Coelho from the Technical University of Kaiserslautern and Martina Maggio from Lund University for their comments on an earlier version of this paper. Their insights and expertise assisted research, however, any errors found are our own.

1 Introduction

Real-time systems used to be closed systems running on specialized hardware. Consequently, security had been given little thought, as no access from the outside to these systems was assumed. However, recent trends show the reuse of more and more components for real-time systems, e.g. the shift from federated avionics architectures to IMA (Integrated Modular Avionics) [24], a growing need for connectivity, especially in the area of IoT and networked systems, and a shift from single to multi- and manycore architectures. These trends lead to an increase in the complexity of real-time systems in general and in particular at the real-time application level. This increased complexity implies that real-time systems cannot be considered closed and inaccessible anymore but instead demands anticipating more vulnerabilities and in turn an increased risk of compromise. Security has to be considered during system design and deployment to prevent unauthorized information disclosure and exploitation of vulnerabilities by a potentially malicious, safety-threatening attacker [22]. This is especially true for systems in safety-critical environments, where real-time time-triggered systems are often used. Research on security in the real-time domain, especially for time-triggered systems, however, is still in its infancy [23].

Time-triggered real-time systems [12] provide highly predictable scheduling behavior to meet strict timing constraints. While real-time online scheduling provides *predictability*, i.e, guarantees that deadlines will be met, but not exact times of execution, time-triggered systems provide *determinism*, i.e., given schedule and time, the task executing is known. However, the very properties of determinism, periodicity, and timeliness can be exploited by an attacker. Reusing complex components in a networked environment inherits all classical security concerns and requires appropriate countermeasures. However, in addition, real-time systems enable a class of attacks specifically targeting the timing of applications and thereby the safety to which these tasks contribute. Security is therefore of high concern for safety-critical real-time time-triggered systems.

Having compromised a large enough set of co-scheduled non real-time or low safety-critical tasks, an attacker can make use of leaked scheduling-related information to fine-tune the compromised tasks' behavior such that they generate maximum interference on subsequently executing victim tasks. In order to stay undetected, an attacker could continue normal operation of the compromised tasks up to the point in time when one of the tasks is executed immediately before a safety-critical task. At this time, the compromised task exploits all of its accessible resources to create an access pattern that maximizes interference on the safety-critical task. For example, writing all accessible memory instead of just the locations accessed when executing as analyzed may result in a cache and/ or DRAM access pattern that maximizes cache-related delays of the subsequently executed safety-critical task. Alternatively, on a multicore system, the compromised task could issue the maximum number of allowed memory requests. If memory requests are not handled properly, this may lead to a deadline miss on another core competing for memory access.

Tools analyzing only the legitimate task behavior to determine, e.g., cache-related preemption delays, are not aware of such malicious behavior. Unless the system designer anticipates maximum preemption delays for all tasks, real-time schedules remain susceptible to such attacks. Furthermore, due to its predictability, time-triggered scheduling is inherently

vulnerable to timing inference based attacks [25]. In this paper, we analyze inference-based vulnerabilities of time-triggered systems and investigate strategies to mitigate attacks based on exploiting these vulnerabilities without violating the very properties that make time-triggered systems attractive to system designers: timeliness and determinism.

Related Work. In literature, several security solutions for real-time systems exist. For example, Völp et al. [21] 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 coexist in the same time window on the same processor. In [16], 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 [21] and [16], we do not focus on preventing timing channels or information leakage. In fact, we assume timing information may be infered.

Yoon et al. [25] introduce a schedule randomization protocol for task sets scheduled under Rate Monotonic which provides obfuscation against timing inference attacks. As long as deadline constraints are not violated, the next task is picked randomly from the ready queue. Each task has a defined budget of tolerated priority inversions which do not violate the tasks deadline constraints. In Section 3, we follow a similar approach for time-triggered systems.

Two examples for state-of-the-art research deal with security for time-triggered communication. In [19], 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. [22] investigate the security of time-triggered transmission channels and shows 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.

Wasicek [23] further presents a threat model for real-time systems, explores security and dependability in the Time-Triggered Architecture (TTA) in great depth and investigates how to enhance TTA for security. In contrast, we do not focus on a specific architecture for time-triggered systems. More precisely, we show how to mitigate directed attacks by randomizing or changing the schedule without violating the timing constraints of time-triggered schedules.

Contributions.

- We analyze vulnerabilities of time-triggered systems with regard to timing inference and malicious behaviour, and show possible attacks which exploit these vulnerabilities.
- We present two practical mitigation strategies for timing inference based attacks with low implementation complexity: an online job randomization algorithm which is able to preserve the timeliness and predictability properties of time-triggered systems, and offline schedule-diversification.
- We evaluate these mitigation strategies with a real-world case study to show that they have low runtime overhead and are practical.

Paper Structure. The remainder of this paper is organized as follows: In Section 2, we present the vulnerability analysis of time-triggered systems against directed timing inference

based attacks. In Section 3, we explore mitigation strategies for directed timing attacks, and evaluate them in Section 4 with a real-world case study.

2 Threat Model and Vulnerabilities

In this section, we first describe our threat and system 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.

2.1 Threat and System Model

For our vulnerability analysis we assume a time-triggered real-time system running on a single core or a single partition with an offline constructed schedule, e.g. in the form of a table. We assume the schedule has been validated and precautions (such as authenticated boot) are in place to ensure that the validated schedule is correctly deployed to the real-time system.

We assume attackers are able to successfully infiltrate the system through undetected vulnerabilities and will eventually exploit infiltrated outposts to attack further parts of the system. Less stringent evaluation requirements make non real-time tasks and low safety-critical tasks primary targets, but we also do not preclude penetrations of higher-critical tasks. Our concern is that attackers exploit these infiltration points to collect timing information about the system and to coordinate subsequent directed attacks against critical, replicated tasks. In particular, we assume that most critical tasks are sufficiently shielded against direct attacks to require 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, 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 that continue normal operation of the compromised tasks until these tasks are executed in a manner where a directed attack 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 continuing denial of service [23], or worse, unsafe control decisions.

The timing information required for coordinating such a stealthy attack can be infered via side channels constructed using shared resources like cache or memory, or through covert timing channels, such as the scheduling-covert-channel described by Boucher et al. [1].

While there exist mitigation strategies for closing side channels (for example in the real-time context, the works of Völp et al. [21] or Mohan et al. [16] on fixed-priority schedulers), they 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 [8]. Meltdown [15] and Spectre [10] are recent examples demonstrating the difficulty of identifying and closing such channels in sufficiently complex architectures. Exploiting non-architectural channels (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, we cannot exclude an integration of cores of this complexity in a real-time system on chip, e.g. for meeting the extended demand of autonomous driving functionalities. Fixing the security flaws of Meltdown and Spectre

results in up to 21 percent performance decrease for Intel client systems [3] and up 25 percent for Intel data center systems [2].

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 this possibility.

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 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. [21]. 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 read the offline defined 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 2.2, 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 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 be 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 behaviour of executing tasks. Access controls and partitioning techniques like cache [14] or bank coloring [26] further constrain the attacker. However, each such countermeasure negatively impacts system performance. Moreover, as we show in greater detail in Section 2.2, mitigating attacks may require cancelling tempting optimizations such as bounding the delay a task can impose through the cache by evaluating their execution patterns. Designers may be tempted to implement optimizations for the sake of increasing performance while neglecting security.

2.2 Vulnerabilities

One of the main vulnerabilities of a time-triggered system lies in its *deterministic behaviour*. 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 4.4, 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 infered 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 behaviour into account. WCET estimated through simulation of the expected behaviour of the system does not account for malicious behaviour. If a task is infiltrated at runtime and, for example, starts accessing the cache to create maximum interference for the next task execution, the tasks simulated worst case does not account for this malicious behaviour if this behaviour 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 [9], [5]. The assumption of cold caches would nullify the described attack of delaying a task through cache misses. We have to choose WCET estimates in a way that they also account for malicious behaviour and we have to check the impact of performance optimizations on security.

In the next section, we show mitigation strategies for directed attacks which prevent an attacker from exploiting the vulnerability that results if malicious behaviour has not been taken into account.

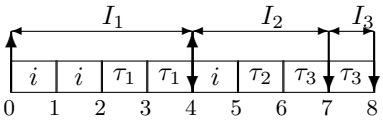
3 Mitigation Strategies

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. This approach is an extended version of the work presented in [13]. The second strategy comprises a set of offline precomputed schedules one of which is randomly chosen at the end of each hyperperiod.

3.1 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 [4], 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. 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



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.

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 [6]. We adopt this concept to guarantee task execution within their respective execution windows even though the scheduling decision is randomized.

3.1.1 Background

Slot shifting uses a discrete time model [11], where the time interval which separates two successive events (i.e. the granularity of the system) is called a slot [18]. 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 capacity interval:

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

The end of the capacity interval is determined by the common deadline of all $\tau_i \in I_j$. If needed, empty capacity intervals without assigned jobs are created to fill gaps between capacity intervals with assigned jobs. 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 3.1.3. In the schedule on the left side of Figure 1, i denotes the idle task.

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 1, a capacity interval is not allowed to start before the end of the previous interval. Note that capacity intervals do not overlap, while execution windows may. Thus, I_2 starts at 4 and ends at the deadline of its assigned set of jobs $\{\tau_2\}$, which is 7. We create interval I_3 accordingly. We show the resulting capacity intervals 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 2 below.

$$sc(I_j) = |I_j| - \sum_{\tau_i \in I_j} C_i + min(sc(I_{j+1}), 0) \tag{2}$$

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 3.1.3, where we present the spare capacity calculation.

If we have calculated all spare capacities, the first capacity interval has a non-negative spare capacity provided the task set is feasible, i.e. its utilization is equal to or less than one since we consider single core systems. Positive spare capacities represent the amount of unused resources and leeway [6] 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 [6] and [18] 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 intervals and their respective spare capacities. For our online randomizing scheduler, we update the spare capacities at runtime to keep track of scheduling decisions.

3.1.2 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

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

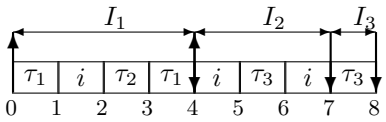


Figure 2 Left: Spare Capacities of I_1, I_2 and I_3 over time, Right: Randomized Schedule.

execution. That is, in the case of zero leeway, the online scheduler randomly selects among the jobs of the current capacity interval. After running a job, we update spare capacities as shown in Section 3.1.1.

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 [20]. 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 [7].

3.1.3 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, here only 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 $t = 0$.

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 are assigned to I_1. At time $t = 4$, τ_3 becomes active and is selected to execute at time $t = 5$ after picking the idle thread to $t = 4$. 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 spare capacity updates in Figure 2.

3.2 Offline Schedule-Diversification

The second mitigation strategy we investigate in this work constructs multiple offline precomputed schedules and switches between them at hyperperiod boundaries. 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 deduce the schedule, as illustrated for example in [16], 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 launch 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 launch a directed attack.

We show in Section 4.5 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. Carefully created execution windows solve deadline and precedence constraints.

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.

Schedule Entropy. Another criterium for schedule selection is schedule entropy as presented in [25]. This measure makes use of the Shannon entropy and is used to quantify the difference, i.e. randomness, between schedules. A subset of feasible schedules is chosen in a way to maximize the schedule entropy for the number of chosen schedules. However, Yoon et al. [25] showed that calculating the schedule entropy has asymptotic exponential complexity because it requires the enumeration of all possible schedules. They provide an approximation of the schedule entropy over the sum of slot entropies called upper-approximated schedule entropy, which is calculated using the probability mass function of a task appearing at a certain slot

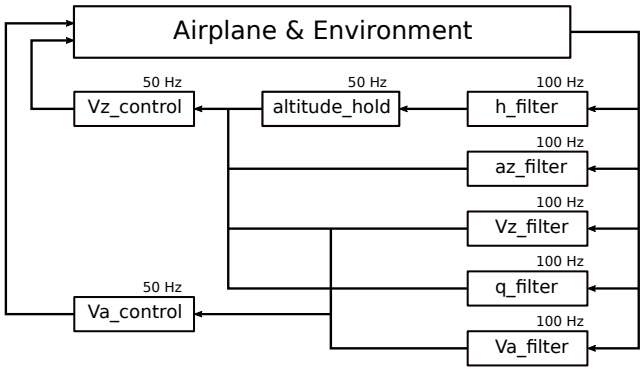


Figure 3 Longitudinal flight controller design.

in the schedule. However, finding a subset with n schedules with the global maximal schedule entropy for all subsets of size n also requires enumeration of all feasible schedules, which is impractical. Therefore, we can apply heuristics for local maxima or select schedules such that the entropy is above a tolerable threshold. For example, we first construct a subset of randomly chosen, feasible schedules with size significantly greater than n, from which we then select the smaller subset of size n with the highest entropy.

4 Evaluation

We evaluated our two directed attack mitigation strategies, which we presented in Section 3, with the ROSACE case study [17]. ROSACE is a practical, real-world example of a real-time system in a safety-critical environment: avionics. This section presents our results.

4.1 Case study: Flight Controller

Pagetti et al. [17] carried out a case study of a longitudinal flight controller of an aircraft. The longitudinal flight controller helps the pilot to accurately track altitude, vertical speed and airspeed of the aircraft. Pagetti et al. describe two control loops: the $Va\_control$ loop handles airspeed control by maintaining the desired airspeed Va; the second control loop — altitude control — combines $altitude\_hold$ and $Vz\_control$. First, $altitude\_hold$ translates altitude commands to vertical speed commands. Then, $Vz\_control$ tracks the vertical speed Vz of the aircraft. Both control loops are fed with filtered data: h, az and q for altitude, vertical acceleration and pitch rate, respectively. Vertical Vz and true airspeed Va are also inputs to the control loops. We show the design of the controller in Figure 3.

According to Pagetti et al. [17], 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.

Table 1 Flight controller task set[17].

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.

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

We construct the execution windows of all tasks from the task set in Table 1. Schorr [18] 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.

4.2 Runtime Overhead for Slot-Level Randomization

Our slot-level randomization algorithm is based on Schorr's [18] 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 [18], shown in Table 3, we approximate the runtime overhead of slot-level randomization, when executed on the same processor.

Slot-level randomization invokes the same functions to update spare capacities and the ready list. The cost of the function to update spare capacities increases with the number of intervals due to cascaded borrowing in the worst case. However, according to the slot shifting

Table 3 Minimum and maximum runtime overhead for single core execution in ns [18].

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

Table 4 Minimum and maximum runtime overhead for single core execution in ns [18].

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

algorithm as explained in Section 3.1.1, 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:

$$t_{ov,rand,max} = rand_{max} + up_{sc,max} + up_{ready,max} + sel_{max} + ISR_{max} \tag{3}$$

Accordingly, the minimum runtime overhead results in:

$$t_{ov,rand,min} = rand_{min} + up_{sc,min} + up_{ready,min} + sel_{min} + ISR_{min} \tag{4}$$

Using the measurements from Table 3 for equation 3 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 [18]. Keeping in mind that ROSACE uses 6 times faster cores than [18] 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 [18]. Dividing these values by 5 as explained earlier, we approximate the minimum runtime overhead for ROSACE to be $t_{ov,rand,min} = 2,500ns$.

4.3 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 [18], which we show in Table 4.

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

Table 5 Exemplary precomputed time-triggered schedule for ROSACE.

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

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.

$$t_{ov,prec,max} = rand_{max} + sel_{max} + ISR_{max} = 10470ns \tag{5}$$

$$t_{ov,prec,min} = rand_{min} + sel_{min} + ISR_{min} = 6410ns \tag{6}$$

Using the same estimation on the execution time of the randomization function for the ROSACE case study as in Section 4.2, 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.

4.4 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 Table 2 suggests. 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.

Assuming each entry has the size of 1 byte, a single schedule with this information needs $13 * 4 = 52$ bytes of memory.

4.5 Discussion

Slot-level randomization proves to be practical, as the approximated overhead in Section 4.2 shows. In the worst case, slot-level randomization uses less than 3 percent of the slot 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. 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.

Even for side channels with low bandwith as we mentioned in Section 2.2, an attacker might identify a small number of schedules after several minutes or a few hours. 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.

$$\left[\binom{50}{1}\binom{49}{1}\binom{48}{1}\binom{47}{1}\binom{46}{1}\right]^2 \times \binom{90}{1}\binom{89}{1}\binom{88}{1} = 4.56 \times 10^{22} \tag{7}$$

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.

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 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.

5 Conclusion

In this paper we described vulnerabilities of time-triggered systems to timing inference based directed attacks and presented two mitigation strategies. 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 infered by an attacker. 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 deadlines. 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 evaluated both 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 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}. We proved both mitigation strategies to be practical. An attacker could still try to launch undirected attacks, but he or she will be easier to detect this way.

For future work, offline schedulers may be enhanced to consider entropy during schedule creation. Moreover, imperfect randomization leaves a residual side channel. Therefore, we are interested in a simulated attack measuring the influence a compromised task has against its victim using our mitigation strategies and to further examine if there exist attack vectors particularly effective against our approach. Lastly, we intend to integrate our approach into a multicore system with partitioned scheduling.

References

1 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.

2 Intel Corporation. Firmware Updates and Initial Performance Data for Data Center Systems. accessed on 26/01/2017. URL: https://newsroom.intel.com/news/firmware-updates-and-initial-performance-data-for-data-center-systems/.

3 Intel Corporation. Intel Security Issue Update: Initial Performance Data Results for Client Systems. accessed on 26/01/2017. URL: https://newsroom.intel.com/editorials/intel-security-issue-update-initial-performance-data-results-client/.

4 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.

5 Christian Ferdinand and Reinhard Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17(2):131–181, Nov 1999. doi:10.1023/A:1008186323068.

6 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, Dec 1995. doi:10.1109/REAL.1995.495205.

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

8 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.

9 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.

10 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. *ArXiv e-prints*, 2018. arXiv:1801.01203.

11 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, Jun 1992. doi:10.1109/ICDCS.1992.235008.

12 H. Kopetz and G. Grünsteidl. TTP-a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, Jan 1994. doi:10.1109/2.248873.

13 Kristin Krüger, Marcus Völp, and Gerhard Fohler. Improving Security for Time-Triggered Real-Time Systems against Timing Inference Based Attacks by Schedule Obfuscation. In

29th Euromicro Conference on Real-Time Systems (ECRTS 2017), Work-in-Progress Proceedings, pages 4–6, 2017.

14 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, Jun 1997. doi:10.1109/RTTAS.1997.601360.

15 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*, jan 2018. arXiv:1801.01207.

16 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.

17 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. Open Source avionics task set. doi:10.1109/RTAS.2014.6926012.

18 Stefan Schorr. *Adaptive Real-Time Scheduling and Resource Management on Multicore Architectures*. PhD thesis, Technical University of Kaiserslautern, March 2015.

19 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.

20 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.

21 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.

22 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.

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

24 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, Oct 2007. doi:10.1109/DASC.2007.4391842.

25 M. K. Yoon, S. Mohan, C. Y. Chen, and L. Sha. TaskShuffler: A Schedule Randomization Protocol for Obfuscation against Timing Inference Attacks in Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016. doi:10.1109/RTAS.2016.7461362.

26 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.