Improving Security for Time-Triggered Real-Time Systems against Timing Inference Based Attacks by Schedule Obfuscation

Kristin Krüger, Gerhard Fohler
Technische Universität Kaiserslautern, Germany
{krueger, fohler}@eit.uni-kl.de

Marcus Völp
SnT - Université du Luxembourg
marcus.voelp@uni.lu

***Abstract*—Covert timing channels in real-time systems allow adversaries to not only exfiltrate application secrets but also to mount timing inference based attacks. Much effort has been put into improving real-time system predictability with the additional benefit of reducing the former class of confidentiality attacks. However, the more predictable the system behaves, the easier timing inference based attacks become. Time-triggered scheduling is particularly vulnerable to these types of attacks due to offline constructed tables that are scheduled with clock synchronization and OS-timer predictability. In this paper, we obfuscate time-triggered scheduling to complicate timing inference based attacks while maintaining strong protection against exfiltration attacks.**

I. Application domain & Challenge

Time-triggered (TT) real-time systems [1] are often used in safety-critical environments where they provide highly predictable scheduling behavior to meet stringent timing constraints. While online scheduling provides *predictability*, i.e, guarantees that deadlines will be met, but not exact times of execution, TT systems provide *determinism*, i.e., given schedule and time, the task executing is known. Leaking the scheduling information of safety-critical tasks enables adversaries to mount targeted attacks misusing this knowledge to defy detection and jeopardize the timeliness and thereby the safety that these tasks contribute to. Security is thus of high concern for safety-critical systems.

Having compromised a large enough set of non real-time or low safety-critical tasks, an attacker can make use of leaked scheduling-related information to fine-tune its behavior such that the set generates maximum interference on subsequently executing victims. For example, to stay undetected, an adversary could continue normal operation of its compromised tasks up to the point when one of its tasks is executed immediately before a safety-critical task. At this time the compromised task exploits all of its accessible memory to create a cache and memory access pattern that maximizes cache-related delays of the safety-critical task. Naturally, tools analyzing only the legitimate task behavior to determine cache-related preemption delays are blind to such malicious behavior. Short of anticipating maximum preemption delays for all tasks, TT schedules remain susceptible to such attacks. Furthermore, due to its predictability, TT scheduling is inherently vulnerable to timing inference based attacks [2].

In this work, we show how we can use an offline constructed TT schedule to impede timing inference based attacks.

II. Motivation

Research on security in the real-time domain, especially for TT systems, is still in its infancy [3]. Meanwhile, TT real-time systems are used in safety-critical environments. Covert timing channels in these systems risk leakage of critical information which threatens safe system operation. According to [4], the assignment of scheduling priorities and the preemptiveness property of tasks may already produce a covert timing channel. An adversary in the system can use covert timing channels to exfiltrate information and infer when a victim runs and later use this information for more targeted system manipulation. The typical multi-vendor development approach for complex real-time systems can grant adversaries access through vulnerabilities in the software of one of the vendors before shipping or later during deployment. Thorough code analysis to remove all vulnerabilities [5] remains a myth because of the sheer code size and the complexity of real-time systems. Furthermore, common scheduling algorithms like Rate Monotonic or Earliest Deadline First impose an order on jobs, which improves system predictability and, however, eases timing inference based attacks. TT scheduling is even more predictable as it uses an offline defined scheduling table. The transformation of security requirements into scheduling constraints is not always possible for real-time systems and the performance overhead can render security solutions infeasible [6].

We propose an extension to TT scheduling which imposes a random order on jobs and transforms security requirements into scheduling constraints at a low expected runtime overhead while still guaranteeing the original real-time constraints. Randomizing job execution order reduces the likelihood of targeted manipulation hitting their victims, even if sufficient information about job scheduling parameters is leaked. Replicating and scheduling the jobs of each replica in a different random order further reduces the residual risk of attacks being effective because manipulations must now hit a majority of replicas to be successful.

III. Problem statement

We assume a TT real-time system with an offline constructed schedule, e.g., in the form of a table. The global clock is considered adequately protected, i.e., not compromisable. It is a very important resource in TT systems. An adversary has infiltrated the system at the task level and tries to collect timing information of the system through scheduling covert-channels [7] to coordinate subsequent attacks. TT schedules are fixed and thus easier to predict. We extend offline constructed TT schedules to obfuscate the schedule and thus impede timing inference based attacks that require exact knowledge when victims run.

IV. Proposed approach and preliminary results

Schedules for TT systems are typically constructed by an offline scheduler [8] which resolves the real-time constraints to be represented in a scheduling table. Our approach analyzes this 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 as start and its deadline as the end. Slot shifting [9] is a real-time scheduling algorithm which guarantees job execution within its execution window based on scheduling tables. Therefore, we chose to integrate our idea with slot shifting.

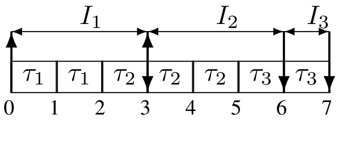
A. Background

Slot shifting uses a discrete time model [10], where the time interval which separates two successive events (i.e. the granularity of the system) is called a slot [11]. Slot shifting consists of an offline and an online phase. In the offline phase, the TT schedule is analyzed to determine available leeway in the schedule. 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 interval. The start of a capacity interval I_j, $start(I_j)$, is defined as the maximum of the earliest start time $est(I_j)$ among its jobs τ_i and the end of the previous capacity interval, see Equations 1 and 2 below.

$$est(I_j) = min(est(\tau_i)) \forall \tau_i \in I_j \tag{1}$$

$$start(I_j) = max(end(I_{j-1}), est(I_j)) \tag{2}$$

The end of the capacity interval is marked by the deadline. 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.



τ_i	est_i	C_i	d_i
τ_1	0	2	3
τ_2	0	3	6
τ_3	3	2	7

Fig. 1: Job set and intervals derived from offline schedule (left)

Three distinct deadlines exist for that job set, thus three capacity intervals have to be created. Starting at time 0, the first interval I_1 starts at 0 and ends at the deadline of its assigned job τ_1, which is 3. The next interval shares the same earliest start time, but according to Equation 2, the capacity interval is not allowed to start before the end of the previous interval. Thus, I_2 starts at 3 and ends at the deadline of its assigned job τ_2, which is 6. Interval I_3 is created accordingly. The resulting capacity intervals are shown 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. It is defined as the interval length minus the worst case execution times C_i of all its jobs τ_i and minus slots borrowed from the succeeding interval, see Equation 3 below.

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

Spare capacities are calculated starting at the last capacity interval. Borrowing occurs in some corner cases, when the current capacity interval provides not enough slots to accommodate all its jobs, which results in a negative spare capacity. Capacity intervals with a negative spare capacity borrow the needed amount of slots from the preceding interval. Negative spare capacities do not equal infeasibility in the scheduling sense. Spare capacities are an abstract way to track "free" slots in a capacity interval.

After all spare capacities have been calculated, 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. Positive spare capacities can be seen as the amount of unused resources and leeway [9] of an interval which can be given to other tasks with overlapping execution windows. Note that capacity intervals do not overlap, while execution windows may.

B. Schedule Obfuscation

At runtime, at the beginning of each slot, the online scheduler is invoked to select the next job. Here we apply our approach to obfuscate the TT schedule. We apply a new scheduling algorithm which selects the next job of the tasks in the ready queue at random. Each job has the same probability of getting chosen for this slot. At the same time, we guarantee timing constraints by running an online acceptance test after a job was chosen. If the job passes the acceptance test, its execution will not cause another job to miss its deadline and it is allowed to run. If it fails the acceptance test, another job of the tasks in the ready queue is chosen until a job passes the acceptance test.

Originally, the online acceptance test in slot shifting is used to integrate aperiodic, i.e., event-triggered tasks into a pure TT system. For now, we do not consider aperiodic tasks in the system, but we integrate a modified version of the acceptance test within our scheduler to enforce timing constraints. Our job acceptance test recalculates the remaining spare capacity in the current capacity interval, as if the randomly chosen job would have been executed. As long as the spare capacity of the current capacity interval remains non-negative, the chosen job is allowed to execute. Otherwise, only a job belonging to the current capacity interval will be accepted.

In order to recalculate spare capacities, we consider three different cases. In the first case, no job was executed. The processor ran idle and the spare capacity of the current capacity interval is decreased by one. In the second case, a job belonging to the current capacity interval was executed, which means the spare capacity of the current capacity interval does not change. In the third case, a job not belonging to the current capacity interval was executed, i.e. prior to its own capacity interval, and thus the spare capacity of the current capacity interval is decreased by one. If this job belonged to a capacity interval borrowing slots from the current capacity interval, the spare capacity of the borrowing capacity interval is increased by one and, as a result, the spare capacity of the current capacity interval is increased by one too, as it lends one slot less to another capacity interval.

C. Example

We will show how the scheduler works for our example jobset 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 WCET of its assigned job τ_3, which results in a spare capacity of -1. I_2 has an interval length of 3, from which the WCET of τ_2 and the slots borrowed by the preceding interval I_3 are substracted, which results in a spare capacity of -1, too. The spare capacity of I_1 is calculated accordingly. Table I shows these spare capacities at time t=0.

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

TABLE I: Spare capacities of I_1, I_2 and I_3 over time

At runtime, the scheduler randomly picks τ_1 for the first slot from the list of ready jobs τ_1 and τ_2. The online acceptance check precalculates the spare capacity for the next slot. As τ_1 executes within its own interval, the spare capacities do not change and τ_1 is allowed to execute. τ_2 gets randomly chosen for the next slot and the check is invoked once more. τ_2 does not execute within its own capacity interval, therefore $sc(I_1)$ is reduced by one. Meanwhile, the capacity interval of τ_2, I_2, borrows one slot from I_1, which means its own spare capacity $sc(I_2)$ is increased by one. Thus, it needs to borrow one slot less from I_1 and therefore $sc(I_1)$ is increased by one, too. In summary, $sc(I_1)$ stays at 0, $sc(I_2)$ is increased by one and τ_2 is accepted for the second slot. If the scheduler would choose τ_2 again for the third slot, $sc(I_1)$ would become negative. Thus, τ_2 is rejected and τ_1 is allowed to run instead and finishes at its deadline. At time 3, τ_3 becomes active. Further scheduling decisions and spare capacity updates are reflected in Figure 2 and Table I.

Combining TT scheduling with our schedule obfuscation method, we impede online predictions about the schedule. Offline predictions are not possible, because the scheduler randomly decides on the next job at runtime. Furthermore, TT scheduling inherently confines application-level leakage

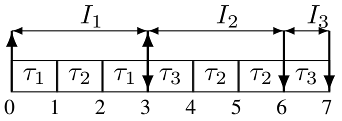


Fig. 2: Randomized schedule

to shared resources which are held across slots [12]. An investigation of leakage countermeasures for such resources is out of the scope of this paper. While randomization does not allow slot-level determinism typical for TT systems, the acceptance test still allows for execution window determinism [13].

V. Envisioned solution

We propose an extension to time-triggered scheduling which strengthens system security against timing inference based attacks. Randomized job selection impedes predictions of the schedule while an online acceptance check guarantees real-time constraints. Execution windows of jobs can be tailored to meet flexibility or determinism requirements, if needed. For example, decreasing the execution window size forces a job to be executed within a smaller time frame and reduces jitter.

For future work, we intend to measure the runtime overhead and evaluate our approach against statistical timing analysis.

References

[1] H. Kopetz and G. Grünsteidl, "TTP-a protocol for fault-tolerant real-time systems," *Computer*, vol. 27, no. 1, pp. 14–23, Jan 1994.

[2] 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)*, April 2016.

[3] A. R. Wasicek, "Security in Time-Triggered Systems," Ph.D. dissertation, Technische Universität Wien, 2011.

[4] J. Son and J. Alves-Foss, "Covert Timing Channel Analysis of Rate Monotonic Real-Time Scheduling Algorithm in MLS Systems," in *2006 IEEE Information Assurance Workshop*, June 2006, pp. 361–368.

[5] C.-Y. Chen, A. Ghassami, S. Nagy, M.-K. Yoon, S. Mohan, N. Kiyavash, R. B. Bobba, and R. Pellizzoni, "Schedule-Based Side-Channel Attack in Fixed-Priority Real-time Systems," University of Illinois, Tech. Rep., 2015.

[6] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. B. Bobba, "Integrating security constraints into fixed priority real-time schedulers," *Real-Time Systems*, pp. 1–31, 2016.

[7] P. K. Boucher, R. K. Clark, I. B. Greenberg, E. D. Jensen, and D. M. Wells, *Toward a Multilevel-Secure, Best-Effort Real-Time Scheduler*. Vienna: Springer Vienna, 1995, pp. 49–68.

[8] S. S. Craciunas and R. S. Oliver, "SMT-based Task- and Network-level Static Schedule Generation for Time-Triggered Networked Systems," in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: ACM, 2014, pp. 45:45–45:54.

[9] G. Fohler, "Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems," in *Proceedings 16th IEEE Real-Time Systems Symposium*, Dec 1995, pp. 152–161.

[10] H. Kopetz, "Sparse time versus dense time in distributed real-time systems," in *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, Jun 1992, pp. 460–467.

[11] S. Schorr, "Adaptive Real-Time Scheduling and Resource Management on Multicore Architectures," Ph.D. dissertation, Technical University of Kaiserslautern, March 2015.

[12] 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.

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