Total Execution Order in Fault-Tolerant Real-Time Systems

Amin Naghavi
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg
Luxembourg
amin.naghavi@uni.lu

Nicolas Navet
Department of Computer Science, University of Luxembourg
Luxembourg
nicolas.navet@uni.lu

Abstract

Many real-time systems nowadays must not only tolerate accidental faults but also targeted attacks. Typically, techniques such as replication and diversification are used to mask the malicious behavior of compromised nodes behind a healthy majority. This work focuses on the replication of event-triggered real-time systems, where prioritized tasks are scheduled non-preemptively on nodes. In such systems, different execution times of replicated jobs on different nodes may lead to their different execution order and different state transitions on nodes. Total order protocols can be used to coordinate nodes to execute jobs in the same order. Previously published total order approaches do not meet all the requirements of real-time systems as a malicious node can inject priority inversion on other nodes in such a way that healthy nodes can no longer guarantee the timely completion of jobs. In this paper, we propose a novel coordination algorithm to detect and tolerate such attacks. In our approach, once jobs are inserted into the ready queues, nodes can proceed with their execution within a bound without further communication until the next release. This bound is updated over time, allowing more jobs from the ready queue to be executed. Upon task release, nodes use reliable communication to share their progress, so they insert the released jobs in the same position in their queues. Nodes evaluate each other's progress before inserting jobs to verify that scheduling bounds have been respected and to detect any priority inversion injection attacks. We evaluate our approach and show that it can guarantee the schedulability of more task sets than other published total order protocols and exhibit low average response times at reasonable run-time overheads.

CCS Concepts

• **Computer systems organization → Real-time systems**; **Redundancy**.

Keywords

Real-Time Systems, Event-Triggered, Replication, Total Order

RTNS 2024, November 06–08, 2024, Porto, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1724-6/24/11
https://doi.org/10.1145/3696355.3699704

ACM Reference Format:
Amin Naghavi and Nicolas Navet. 2024. Total Execution Order in Fault-Tolerant Real-Time Systems. In *The 32nd International Conference on Real-Time Networks and Systems (RTNS 2024), November 06–08, 2024, Porto, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3696355.3699704

1 Introduction

Real-time systems in diverse application domains, including industrial automation [28, 30, 43], telecommunications [4], power grid [13, 38], and cooperative driving [2, 12], increasingly face cyberattacks that threaten their safety. Replication, which is already widely applied to ensure safety, can help to tolerate targeted attacks [11, 14, 47], provided that common mode faults are avoided through diversification [3, 51, 55].

In this paper, we focus on event-triggered real-systems and assume all tasks on a node (including the RTOS) are replicated on other nodes. In these systems, diversified versions of replicated tasks may not always behave identically or finish simultaneously on all nodes. This might lead to different execution orders on different nodes. Consider a system with two nodes and three tasks: τ_a, τ_b, and τ_h (highest priority). Jobs for τ_a and τ_b release at time t_0, and τ_h releases at t_1 ($> t_0$). By t_1, the first node has executed τ_a, while the second node has executed both τ_a and τ_b. The first node will insert τ_h after τ_a and execute it before τ_b, whereas the second node will execute τ_h after both τ_a and τ_b.

Our proposal contributes to guaranteeing total execution order in replicated systems. Many applications, such as ADAS automotive systems, have data dependencies between tasks [19, 24, 32]. Consistent job execution order among nodes ensures comparable outputs when tasks are replicated [15]. Additionally, checkpointing and recovery are simpler when all nodes undergo the same state transitions [42]. Rodrigues et al. [52] and Wang et al. [57] propose different protocols to guarantee the total order of prioritized tasks. In Rodrigues' method [52], upon task release, nodes identify the portion of the queue where jobs have been executed by at least one node. The released jobs are then inserted, only after this portion, into the remaining part of the queue according to their priorities. To address the group priority inversion problem in Rodrigues' method, Wang et al. [57] ensure total order by selecting the insertion point of each released job after the section of the queue where the majority of nodes have already executed. This approach requires rolling back the execution of any job that nodes executed beyond the insertion point. Later, we illustrate how both methods fail to meet real-time task deadlines on healthy nodes under *priority inversion attacks*, even with a minority of nodes being compromised.

To tolerate priority inversion attacks, one might always force all nodes to remain idle until the WCET of the job before starting the next job in the queue. However, this method (which we call the *Simple* method) is inefficient as nodes often stay idle while there are jobs in their ready queues, especially considering that the WCET of jobs might be very pessimistic.

Our method, similar to Rodrigues' and Wang's methods, allows nodes to execute jobs in their ready queues without coordination before each execution. Coordination is only needed before nodes insert jobs into their ready queues at job release to ensure total order. In our method, the execution order of jobs is based on the progress of the node that finishes jobs earlier than others (the *front-runner*). Therefore, there is no need for rollbacks. Our approach uses a scheduling rule to limit how far nodes can advance without coordinating with others. This rule ensures that jobs executed by a healthy node can have their replicas executed by other nodes, including the *back-runner* (the node that has advanced the least), in the same order, without missing their deadlines. At the release of tasks, nodes communicate their progress and validate each other's progress. Each node identifies the nodes attempting to induce unbounded priority inversion and finds the non-faulty (non-malicious) front-runner that has adhered to the scheduling rules. All nodes insert the released tasks in the same position in their ready queue after all the jobs executed by the healthy front-runner. Any priority inversion caused by the non-faulty front-runner on the back-runner remains within the slack of tasks. Our approach works equally well with any priority scheme, including fixed or dynamic priority assignments.

In the rest of this paper, we discuss related work in Section 2 and present our system model in Section 3. In Section 4, we introduce priority inversion attacks. Our method is explained in Section 5, and the algorithms are detailed in Section 6. Section 7 describes the calculation of slacks under the Rate Monotonic (RM) and the Earliest Deadline First (EDF) scheduling policies. The evaluation of our method is presented in Section 8, and Section 9 provides the conclusion and suggests directions for future research.

2 Related Work

Whereas early works focused primarily on tolerating permanent [44] and transient accidental faults [1, 34, 46], securing real-time systems against cyberattacks gained attention much more recently. For example, Hasan et al. [26] investigate the allocation of security tasks. Nasri et al. [45] analyze conditions for successful time-domain attacks. Li et al. [39] extend the crash-fault tolerant coordination service Zookeeper to include real-time recovery. Yoon et al. [61] and Krüger et al. [37] investigate schedule randomization in event- and time-triggered systems, respectively. Zhang et al. [62] investigate recovery from sensor attacks.

Coordinating the execution order of replicated tasks is essential for cause-effect chains [24] and has been investigated in the context of replicated state machines [54] as well as for primary/backup systems [64]. Rodrigues et al. [52] and Wang et al. [57] extend totally ordered execution to prioritized tasks. In the presence of priority-inversion injection attacks, these methods fail to guarantee timeliness, as we shall see in Section 4.

Gujarati et al. [23] explore replica coordination when using switched Ethernet in the presence of non-malicious Byzantine errors by subdividing processes to tasks and executing each task after periodic execution of the interactive consistency protocol proposed by Pease et al. [48]. Poledna et al. [50] guarantee consistency by ensuring that all replicas deliver the same messages in the same order by introducing timed messages using which, the messages sent from tasks are only delivered after their deadline or their worst-case response time. Fara et al. [18] coordinates distributed voting on nodes based on passive waiting and Logical Execution Time (LET).

Our work ensures the same total execution order of replicated prioritized tasks on all nodes while guaranteeing their timeliness. We assume that nodes communicate through a broadcast channel or protocol that ensures real-time reliable broadcasts. Several works have discussed bounding delays in reliable messaging schemes [31]. Kozhaya et al. investigate real-time reliable broadcast [35] and atomic broadcast [36] for arbitrary networks, while Roth and Haeberlen [53] discuss optimizations for systems with broadcast channels such as buses, where these channels exhibit some of the required reliability properties. Zhang et al. [63], factor out ordering from consensus and use the median of the timestamps assigned by $2f + 1$ out of $3f + 1$ nodes for ordering.

We consider non-preemptive task models that have been investigated in many works [10, 21, 27, 29, 59]. Yao et al.[60] and Bertogna et al.[8] calculate the maximum blocking allowed before executing a task to facilitate schedulability tests for various preemption models and to place preemption points in tasks. We utilize the findings of these two studies, along with insights from Phan et al. [49], to determine the available slack for tasks, taking into account the release overhead. Slack of tasks is used by other state-of-the-art to schedule aperiodic tasks, address self-suspension, or tolerate overrun in mixed-criticality systems [25]. In this work, we use the slack of the tasks to ensure total execution order.

3 System model

System and task model. This work contributes to the coordinated execution of sporadic tasks in fault-tolerant event-triggered real-time systems. We characterize sporadic tasks $\tau_i \in \mathbb{T}$ of the task-set $\mathbb{T} = \{\tau_1, ..., \tau_n\}$ through triples (C_i, T_i, D_i), where C_i is the worst case execution time (WCET), T_i is the minimal inter-release time and $D_i \leq T_i$ is the relative deadline of the task. We assume a non-preemptive scheduling (like Rodrigues [52] and Wang [57]). We consider replication over m nodes $P_1, \ldots, P_m$, each executing all tasks in $\mathbb{T}$ using the same scheduling algorithm. We assume further that all instances of a task, despite being diversified, have an execution time with an upper bound C_i (see also Fellmuth et al. [20]). Each task τ_i emits a sequence of jobs $\tau_{i,j}$, which are released by a corresponding event at the release time $r_{i,j}$. This defines the job's absolute deadline as $d_{i,j} = r_{i,j} + D_i$. We omit job indices and write τ_i for $\tau_{i,j}$ when the index of the job is not important in the context. During run-time, the last release time of a task τ_i is captured by r_i^{prev}. At any time t, the next earliest release of τ_i is denoted by $\rho_i(t)$, which is calculated as $\rho_i(t) = \max(r_i^{prev} + T_i, t)$.

We first introduce our approach for fixed-priority non-preemptive scheduling, before extending it to dynamic-priority scheduling in Section 7. We denote the set of higher and lower priority tasks than τ_i by hp_i and lp_i, respectively.

As usual, we define a schedule as feasible if all jobs $\tau_{i,j}$ of all tasks τ_i receive C_i time in the interval $[r_{i,j}, r_{i,j} + D_i]$. The slack of task τ_i ($slack_i$) represents the maximum allowable delay in the execution of any job of τ_i due to idleness or execution of lower priority jobs without causing it to miss its deadline.

Our goal is to ensure that the schedules remain feasible with our approach using a given priority assignment if the tasks are schedulable using a non-preemptive uniprocessor scheduling.

Execution and communication. It is assumed that each node keeps a queue of jobs that have been released. A *released queue* q_k of node P_k consists of jobs that have been completed and those ready to be executed (the latter is also referred to as the *ready queue*). Each node tracks its *progress* which is an index in the queue representing the number of jobs that the node completed their execution. We denote the progress of a node P_k as $prog_k$. For simplicity, we assume that jobs in the released queue are never removed. However, in real implementations, nodes can remove jobs after execution by the back-runner. In such cases, a node's progress would be the number of removed jobs plus the queue index of the last executed job.

Healthy nodes always insert the released tasks in the same position in their queues, therefore maintaining identical released queues. However, nodes might differ in their progress. When an event occurs, its corresponding task will be generated on every node in a buffer known as the *Released Buffer*(RB_k), from which it will eventually be inserted into the ready queue. We assume that, at each release, each node shares its progress along with other information using a triple ($prog_k$,toc_k,run_k). Here, toc_k represents the time when the node completed the last executed job, and run_k is a single-bit variable where $run_k = 1$ indicates that the node is executing the job at position $prog_k + 1$ in the queue during communication.

We assume that nodes are connected via a network with a bounded transmission time, where messages of a node are handled by a component integrated into its CPU (such as an integrated DMA-based Network Interface Controller [9] or a co-processor [7]). Therefore, a node simply writes the data to its communication interface buffer and then continues executing tasks while the communication occurs in parallel. Communication takes place through a reliable broadcast channel (such as a CAN Bus [53]) capable of delivering messages unchanged within a bounded time to all correct nodes. Alternatively, when using communication channels that do not provide such features (such as switched Ethernet or unicast channels), a real-time reliable broadcast mechanism can be employed [35]. In this case, we assume such broadcast mechanism guarantees detection of equivocation using only $\lfloor \frac{m}{2} \rfloor + 1$ healthy nodes [53, 56]. We assume that all nodes are aware of a *timeout* value representing an upper bound on communication delays.

Synchronization assumption. In this paper, we assume that clocks are synchronized using a clock synchronization protocol [16]. To account for potential clock deviations between nodes, we can either assume that each event is timestamped by a central trusted entity before distribution among nodes—explicitly incorporating clock imprecision into the slack calculation and the estimation of the next earliest release of tasks—or implement a sparse time base as described by Kopetz et al. [33]. For simplicity, we do not explore these options further in this paper and leave them for future work.

Threat model. In this paper, we assume that up to f out of the $m \geq 2f+1$ nodes might be faulty, either failing silently or behaving in a Byzantine manner. In the fail-silent case, a faulty node simply stops providing any further output or progress information. We also consider the node as a failed node if it loses communication with other nodes. If a node fails in a Byzantine way, it acts arbitrarily. Tasks on a Byzantine node might produce incorrect or no outputs, or exceed their WCETs and even miss their deadlines. A Byzantine node might provide false or no progress updates to other nodes, or execute tasks out of the order compared to other nodes.

A node's Byzantine behavior might result from being under the full control of an attacker, from accidental faults, or from faults induced by the attacker. Additionally, an attacker might eavesdrop on the network and send messages on behalf of a compromised node.

We assume that the network is protected from overload by malicious nodes through a traffic management policy, like traffic policing on the ingress ports of Ethernet switches or bus guardians and active stars in the case of shared buses. Therefore, we consider Byzantine nodes cannot disrupt the communication of other healthy nodes over the shared channel beyond the specified timeout.

We assume that tasks on healthy nodes always generate correct outputs and require at most their WCETs to complete their execution. A healthy node always executes scheduling and release algorithms correctly and participates in the communication at each task release.

The correct outputs from at least $f+1$ healthy nodes are required to mask the faulty outputs from at most f faulty nodes. Nevertheless, our protocol guarantees that any arbitrary behavior of faulty nodes does not interfere with the timeliness of tasks on healthy nodes, irrespective of the number of faulty nodes.

4 Priority-Inversion Injection

Priority inversion (PI) occurs when a node executes a lower-priority job while higher-priority jobs are ready to execute. Priority inversion naturally occurs in non-preemptive scheduling when schedulers must finish the currently running job after a higher priority job has been released. In addition, replicated systems for different purposes such as providing total order [57] or scheduling randomization [61] may require schedulers to deviate from their regular schedule. We say priority inversion is *injected* if it is a result of the schedule of other nodes. In such systems, malicious nodes may inject priority inversion and potentially cause the targeted healthy nodes to miss the deadlines of their tasks. We call this a *priority-inversion injection attack*.

In the following, using the example in Figure 1, we demonstrate how priority-inversion injection attacks lead to deadline misses in both Rodrigues' [52] and Wang's [57] total order protocols. This example involves five tasks replicated across three nodes, where τ_a, the highest-priority task, released a job at r_a, while the jobs of the other tasks were released earlier at t_0. In Figure 1a, the malicious node P_1 pretends to have executed the released jobs τ_b, τ_c, τ_d, and τ_e in order to attack Rodrigues' method. Similarly, in Figure 1b, P_1 pretends to have executed τ_b, τ_c, and τ_d to attack Wang's method.

Rodrigues' priority-based totally ordered multicast. In Rodrigues' method [52], when a job is released, nodes lock their ready queues and send their peers the queue of jobs they have executed. Each node inserts the jobs into its ready queue based on job priority but after all the jobs in the received queues from other nodes.

In Rodrigues' method, a single malicious node can perform a priority inversion attack by falsely claiming to have executed enough of the released jobs to cause other nodes to fall behind and miss their task deadlines as they attempt to catch up. As illustrated in Figure 1a, a malicious node P_1 misused Rodrigues' protocol to inject

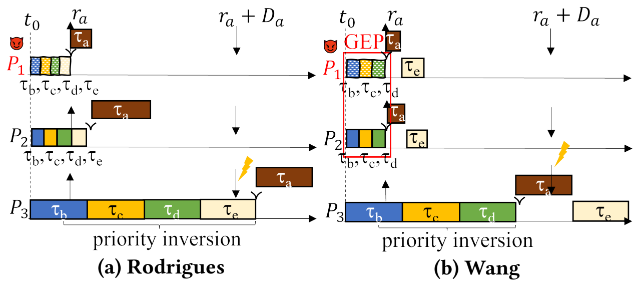


Figure 1: Priority inversion injection attack to Rodrigues' [52] and Wang's [57] approach.

priority inversion into P_2 and P_3. To match P_1's execution order, upon τ_a's release at r_a, both P_2 and P_3 insert τ_a, after the jobs τ_b, τ_c, τ_d, and τ_e. Therefore, P_3 which executes jobs with their WCET, requires up to $C_b + C_c + C_d + C_e$ to complete the jobs executed by P_1 which leaves insufficient time to complete τ_a before its deadline.

In conclusion, while Rodrigues' method guarantees a total order of execution, it is vulnerable to priority inversion injection attacks, which can jeopardize the timeliness of tasks on healthy nodes. Note that plausibility checks verifying whether all jobs in the reported queues are executed for at least their BCET will not resolve the issue, as an attacker can still cause the back-runner to miss the deadlines of its jobs by pretending to execute jobs with their BCETs.

Wang's global execution progress method. In Wang's method [57], a total order module provides each node's scheduler with a queue of jobs to execute until a new release requires reordering this queue. Nodes insert the released jobs based on their priority and after the *Global Execution Progress* (*GEP*). GEP is determined based on the minimum progress of $f+1$ nodes that have progressed ahead of other nodes. The nodes roll back the execution of any jobs that they have executed beyond the insertion point. Wang's approach is vulnerable to priority-inversion injection attacks, where f compromised nodes (out of $2f+1$) advance GEP by leveraging the progress of a faster healthy node. In this case, a higher-priority released task is inserted after the GEP, meaning lagging nodes must execute jobs up to the GEP before they can execute this task. If a node falls far enough behind, this can lead to missed deadlines. This attack is also possible with fewer malicious nodes ($f' < f$) if there is enough gap between the progress of the slowest among the $\lceil m/2 \rceil - f'$ fastest healthy nodes and the slowest node in the system.

The example in Figure 1b shows an attack to the Wang protocol. The malicious node P_1 mimics the progress of the healthy node P_2 to advance the GEP by τ_b, τ_c, and τ_d. Therefore, the lagging node P_3, which experiences execution times close to the jobs' WCETs, first needs to catch up before it can execute the recently released job τ_a. Therefore, τ_a missed its deadline.

Coordination at each scheduling event. An alternative approach involves coordinating before each job execution. Nodes wait for $f+1$ nodes to complete the current job before proceeding with the next, where up to f nodes might be malicious. However, like in Wang's method, faulty nodes could exploit the progress of a fast node, causing healthy nodes to lag behind and miss deadlines.

Nodes may idle after finishing a job until all nodes complete it. However, this approach may significantly slow down progress, both in the presence and absence of malicious nodes, and it introduces additional communication overhead before each job execution.

5 Mitigating PI-injection Attacks

Our approach coordinates the insertion point of jobs in the ready queue at their release without requiring the rollback of job executions. This allows nodes to progress with job execution without waiting for their replicas on other nodes, although nodes might occasionally idle to bound priority inversion on other nodes. Additionally, the method protects against priority-inversion injection attacks by requiring nodes to validate the progress they adopt from others. This method consists of two components: the *scheduling* component and the *release* component. After defining *imminent tasks* and *worst-case projection*, we give an overview of these two components and their roles. Then we illustrate how they prevent a priority-inversion injection attack.

Imminent task: We say task τ_i is *imminent*, if it does not have any job waiting for execution in the ready queue.

Worst-Case Projection: Let $prog_{br}^{prev}$ denote the progress of the healthy back-runner (i.e. the number of jobs the back-runner has executed) known by all nodes after the insertion of the last released job. Additionally, let toc_{br}^{prev} denote the time at which the healthy back-runner begins executing the job at the progress $prog_{br}^{prev} + 1$ in the queue (which typically aligns with the time that it completes the job at $prog_{br}^{prev}$).

A node P_k can calculate the worst-case projection (WCP) of any progress $prog > prog_{br}^{prev}$ in q_k, as:

$$t_w(prog) = toc_{br}^{prev} + \sum_{\tau_i \in q_k[prog_{br}^{prev}+1, prog]} C_i \qquad (1)$$

The worst-case projection of a progress index $prog$ is the latest point in time by which a back-runner must complete all the jobs in q_k up to $prog$, assuming it requires the WCET for completing each job in q_k from $prog_{br}^{prev} + 1$ to $prog$. At run-time, $t_w(prog_k + run_k)$ specifies the latest time by which a healthy back-runner finishes the jobs that P_k completed or started their execution.

Scheduling component. Our scheduling component allows node P_k to progress independently through its ready queue, even though this may cause priority inversion for tasks on lagging nodes. Before executing each job, P_k performs a slack check to ensure that tasks on lagging nodes can still meet their deadlines. It checks whether the slack of higher-priority imminent tasks can accommodate the priority inversion caused on other nodes by executing this job. If the priority inversion doesn't exceed the slacks, P_k proceeds with the execution; otherwise, it idles, waiting for a task release or for the lagging nodes to catch up. In order to make this decision, since P_k is unaware of the back-runner's latest progress, it uses the WCP of its progress and the WCET of the next job in the ready queue to emulate the worst-case behavior of the back-runner.

Release component. Upon a job release, nodes communicate and validate their peers' progress to find the non-faulty front-runner. They verify that the WCP of any progress that P_k has undergone by executing jobs does not exceed the slack bound of higher-priority imminent tasks. If it exceeds, they consider the node as faulty since such a node did not adhere to the bounds enforced by the scheduling component. Nodes then determine the *insertion*

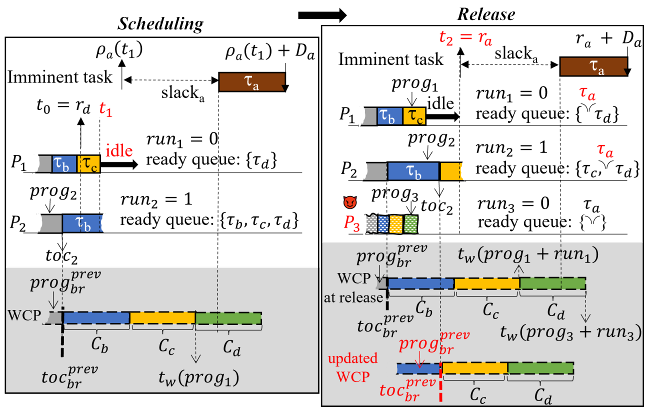


Figure 2: Interplay of scheduling and release algorithms to prevent priority-inversion injection causing deadline misses. P_1 upon running the scheduling algorithm at t_1 remains idle, and at t_2 inserts the released task τ_a in its ready queue.

point offset, which is the progress index of the last job that the non-faulty front-runner has already executed or the job it is currently executing. Therefore, every healthy node that correctly executes the release algorithm inserts the released job after the insertion point offset, considering the priority of the jobs in the ready queue after this offset. Furthermore, nodes find the non-faulty back-runner and update $prog_{br}^{prev}$ and toc_{br}^{prev}. A non-faulty back-runner is a node that has not taken more than the WCET of jobs to finish them and has not been idle while lagging behind other nodes.

Illustrative Example. Figure 2 shows how our scheduling and release components interact by maintaining and validating each node's progress. It illustrates the function of these components on the node P_1 in a system with 3 nodes, where node P_3 tries to induce priority inversion attack. We also illustrated the back-runner node P_2 to show how the variables $prog_{br}^{prev}$ and toc_{br}^{prev} are assigned.

The left-hand side illustrates the scheduling decision on P_1 at time t_1, before which task τ_d was released at t_0. The variables toc_{br}^{prev} and $prog_{br}^{prev}$ were set at t_0 based on the P_2's progress ($prog_2$) and the time P_2 made that progress (toc_2). At time t_1, node P_1, has completed lower priority jobs τ_b and τ_c while the high priority task τ_a is imminent. Before starting the execution of τ_d, node P_1, compares the earliest release plus slack of higher priority imminent task τ_a with the t_w of its progress plus C_d. $t_w(prog_1)$ is calculated by adding the WCET of all of the jobs that P_2 executed after $prog_{br}^{prev}$ to toc_{br}^{prev}. Since $t_w(prog_1) + C_d$ is larger than the slack of the higher priority imminent task τ_a, P_1 remains idle at t_1.

The right-hand side of Figure 2 shows the release of τ_a at time t_2 when nodes coordinate by exchanging the progress they made. P_3 claims to have completed τ_d and jobs in the released queue before that, while P_2 reports that it has completed τ_b and is executing τ_c. The node P_1 validates the progress of the front-runner by calculating the worst-case projection of each progress until $prog_3 + run_3$. Since $t_w(prog_3 + run_3)$ exceeds the slack of τ_a, P_1 considers P_3 as faulty and discards its progress. Hence, P_1 identifies itself as the healthy front-runner as it already finished τ_c. Since P_1 is idle, it inserts the higher-priority job τ_a after τ_c, but before the lower-priority job τ_d, in its ready queue. P_2, like P_1, detects the faulty node P_3, identifies P_1 as the non-faulty front-runner, and inserts τ_a after τ_c. Finally, both nodes P_1 and P_2, update $prog_{br}^{prev}$ and toc_{br}^{prev} based on the progress of the non-faulty back-runner P_2. In the next section, we detail each step of our release and scheduling algorithms.

6 Resilient Insertion Point Algorithm

In this section, we explain our release and scheduling algorithms that guarantee the timeliness of replicated tasks and total order.

Release Algorithm. We present in Algorithms 1 and 2 our release algorithm which is resilient to priority inversion injection attacks. Algorithm 1 is executed on each node P_k when nodes receive an event. This algorithm might initiate the communication by sending P_k's progress to other nodes. Algorithm 2 is executed upon this communication timeout when nodes have already received the progress updates from other nodes. Each of these algorithms runs non-preemptively until completion or until it exits. The algorithms running on a node P_k use the state variables updated in the last execution of the release algorithm, as well as the released buffer RB_k. One of these variables is the last insertion point offset (ipo^{prev}). ipo^{prev} is calculated based on the progress of the non-faulty front-runner from the previous execution of the release algorithm. The execution order of jobs before ipo^{prev} cannot be changed, as it is assumed that a non-faulty node has already executed them, and released jobs are always inserted after this point. The other inputs are $prog_{br}^{prev}$ and toc_{br}^{prev} which have been assigned in the previous release. By executing the release algorithm, nodes find the new insertion point offset (ipo), the progress of the back-runner ($prog_{br}$), and the time that the back-runner starts its next execution (toc_{br}).

Release Algorithm (executing at release): First, when a job τ_e releases, the Algorithm 1 on each node P_k activates and adds the job to the node's released buffer RB_k (Line 1). If there are other jobs than τ_e in RB_k (Line 2), it means that nodes are already in the process of communicating their progress to insert another released job in their queues. Therefore, no more action is needed (Line 3), and after the *timeout* of that communication, the node using Algorithm 2 inserts all released jobs in the RB_k in its ready queue. *timeout* is the upper bound for communication delay, assigned as a constant value to all nodes at design time.

If using Algorithm 1, P_k realizes that all nodes can complete all the released jobs by $r_e + timeout$, it can immediately determine the insertion point of the released job without initiating communication. This check is done by every node independently using WCP. If the t_w of the last progress in the released queue (calculated based on Equation 1) is less than $r_e + timeout$ (Line 4), it implies that even on a node where jobs require their WCET to complete, all released jobs would be finished by $r_e + timeout$; therefore, all healthy nodes would complete these jobs. In this case, P_k can consider the size of the released queue ($|q_k|$) both as the new insertion point offset and as the progress of the back-runner (Line 5). It also considers the maximum of r_e and t_w of the final progress in q_k (i.e. $|q_k|$) as the toc_{br} to be used to calculate t_w in the next execution of the algorithm (Line 6). Then the node inserts the jobs in RB_k into its ready queue (Line 7). It also updates ipo^{prev}, $prog_{br}^{prev}$, and toc_{br}^{prev} based on the new values and update the last release time (r_i^{prev}) of each job (τ_i) in RB_k to its latest release time (Line 8).

Algorithm 1: Release algorithm (executing at release event)

```
Event     : on event $e$ at $r_e$, releasing $\tau_e$ on node $P_k$
Variables : $ipo^{prev}, prog_{br}^{prev}, toc_{br}^{prev}, RB_k, LockProg$
1  $RB_k$.append($\tau_e$)
2  if $|RB_k| > 1$ // if nodes are communicating to insert other jobs
3      exit()
4  if $prog_k + run_k = |q_k|$ and $t_w(|q_k|) \leq r_e + timeout$ // the
   // slowest back-runner finishes released jobs by $r_e + timeout$
5      $ipo = prog_{br} = |q_k|$
6      $toc_{br} = \max(r_e, t_w(|q_k|))$
7      insertByPriority($q_k$,$RB_k$,$ipo$)
8      updatePrev($ipo, prog_{br}, toc_{br}, RB_k$)
9  else
10     if $t_w(prog_k + run_k) > r_e + timeout$ // enough dynamic slack
       // for idling after current execution to finish communication
11         $LockProg = prog_k$
12         sendToAll($prog_k, toc_k, run_k$)
13     else // allowing executing next job after current execution
14         $LockProg = prog_k + 1$
15         $\tau_{next_k} = q_k[prog_k + 1]$
16         sendToAll($prog_k + 1, toc_k + C_{next_k}, 1$)
17     setTimer($r_e + timeout$)
```

If nodes are unaware of each other's progress until $r_e + timeout$, they communicate their progress to other nodes. Nodes lock their progress by setting the $LockProg$, before communicating that to other nodes to avoid the necessity of rollbacks. When a node locks its progress, the progress of the node can never exceed the locked index. A node P_k that is already executing a job can continue with the execution of the job during communication, and it only idles during communication, after finishing the execution of its running job when there is enough dynamic slack for the communication. Dynamic slacks are available when jobs have been executed earlier than their WCETs. If the $t_w(prog_k{+}run_k)$ is bigger than $r_e{+}timeout$, it means that a hypothetical back-runner that executes jobs between $prog_{br}^{prev} + 1$ and $prog_k + run_k$ with WCET would finish them after $r_e{+}timeout$ (Line 10). Therefore, P_k executed jobs earlier than their WCETs and there is dynamic slack until $r_e{+}timeout$. Thus, the node P_k locks its progress from $prog_k$ and sends its progress information (Lines 11-12). If there is not enough dynamic slack available until $r_e + timeout$, the node locks its progress from $prog_k + 1$ and sends $prog_k{+}1$ as its progress (Lines 14-16). This allows a lagging node to execute the next job without remaining idle during communication. As the completion time of the next progress ($prog_k + 1$) is unknown, the node adds the WCET of the next job in its ready queue (τ_{next}) to toc_k. It then broadcasts this, along with $prog_k + 1$ and run_k = 1 to notify that it will start executing the job after τ_{next} before the timeout. After sending its progress, node P_k sets a timer for $r_e{+}timeout$, by which time it expects to receive progress information from all other nodes (Line 17). When the timer expires, the node executes Algorithm 2, which we explain next.

Release Algorithm (executing at timeout): After receiving all the information sent at r_e by other nodes, the node stores this data, along with its own progress, in a set called $DATA$. Algorithm 2,

Algorithm 2: Release algorithm (executing at timeout)

```
Event     : on timeout trigger at $r_e + timeout$
Variables : $ipo^{prev}, prog_{br}^{prev}, toc_{br}^{prev}, RB_k, DATA, LockProg$
1  if $isTrue(\forall(prog_j, toc_j, run_j) \in DATA, prog_j = |q_k|)$:
   // all nodes have finished all the released jobs
2      $ipo = prog_{br} = |q_k|$
3      $toc_{br} = r_e + timeout$
4  else
5      $ipo = unknown; prog_{br} = unknown$
6      $prog_i = \max_{\forall(prog_j,toc_j,run_j)\in DATA}(prog_j)$ // front-runner
7      $run_i = \max_{\forall(prog_j,toc_j,run_j)\in DATA | prog_j = prog_i}(run_j)$
8      if $prog_i + run_i \leq ipo^{prev}$ : // prevent ipo from decreasing
9          $ipo = ipo^{prev}$
10     else
11         while $ipo = unknown$: // finding non-faulty front-runner
12             $wpr = t_w(prog_{br})$
13             for $\tau_a \in q_k[prog_{br} + 1, prog_i + run_i]$:
14                 if $isTrue(\exists\tau_h \in hp_a, wpr + C_a > \rho_h(r_e) + slack_h)$:
                   // priority inversion on a back-runner finishing jobs
                   // until $prog_i + run_i$ may exceed slack of imminent tasks
15                     $DATA.remove(prog_i, \_, run_i)$
16                     $prog_i = invalid$; break // front-runner is faulty
17                 $wpr = wpr + C_a$
18             if $prog_i \neq invalid$: // non-faulty front-runner is found
19                 $ipo = prog_i + run_i$
20 insertByPriority($q_k$,$RB_k$,$ipo$) // insert jobs in $RB_k$ after $ipo$
21 while $prog_{br} = unknown$: // finding non-faulty back-runner
22     $prog_i = \min_{\forall(prog_j,toc_j,run_j)\in DATA}(prog_j)$ // back-runner
23     $toc_i = \max_{\forall(prog_j,toc_j,run_j)\in DATA | prog_j = prog_i}(toc_j)$
24     $\tau_{next_i} = q_k[prog_i + 1]$
25     if $r_e < toc_i + C_{next_i} \leq t_w(prog_i) + C_{next_i}$: // back-runner
       // was not idle since last release and did not exceed WCETs
26         $prog_{br} = \max(prog_i, prog_{br}^{prev})$ // non-faulty back-runner
27         $toc_{br} = \max(toc_i, toc_{br}^{prev})$
28     else
29         $DATA$.remove($prog_i$,$toc_i$,_) // back-runner is faulty
30 updatePrev($ipo, prog_{br}, toc_{br}, RB_k$) // update state variables
31 $LockProg = \infty$ // unlock the progress
```

which is executed on P_k at $r_e{+}timeout$, uses $DATA$ to determine ipo and the insertion point of the released jobs based on the progress of the healthy front-runner. It also assigns $prog_{br}$ and toc_{br} based on the information it received from the healthy back-runner.

If $DATA$ shows that all nodes completed all jobs in their released queue, then the node P_k can consider $|q_k|$ as both insertion point offset ipo and the progress of the back-runner $prog_{br}$ (Lines 1-2). In this case, since no job is executing on any node, the node considers $r_e + timeout$ as toc_{br} (Line 3). This is because the back-runner starts executing its next job at $r_e{+}timeout$ immediately upon being inserted in the ready queue. If $DATA$ contains a node that has not completed all jobs in its ready queue by r_e, the algorithm sets both ipo and $prog_{br}$ to $unknown$ (Line 5) to be determined later.

To ensure that released tasks are inserted after any job executed by any node, the ipo value must always increase. Therefore, if the previous insertion point offset (ipo^{prev}) exceeds the progress of the last job that the front-runner finished its execution or is currently executing, the algorithm does not update the ipo (Lines 8-9). Otherwise, node P_k determines ipo after identifying the non-faulty front-runner (Line 11). First, the node checks whether t_w of any intermediate progress that the front-runner has made (i.e. until $prog_i + run_i$) would exceed the $\rho_h(r_e) + slack_h$ of any higher priority imminent task τ_h (Lines 12-14). If it exceeds, the front-runner will be considered faulty, as it has not complied with the scheduling rule, and the next front-runner will be evaluated (Lines 15-16). Otherwise, the non-faulty front-runner has been found and $prog_i + run_i$ will be considered as the insertion point offset (Lines 18-19). Subsequently, the jobs in the RB_k will be inserted in the released queue q_k based on their priority after ipo (Line 20).

The node P_k finds the progress of the non-faulty back-runner in $DATA$, if $prog_{br}$ has not yet been assigned (Line 21). It first examines the progress of the node that has achieved the minimum progress, and latest among other nodes with the same progress (Lines 22-23). The progress of a back-runner P_i is not faulty if the $t_w(prog_i)$ plus C_{next}, which is the WCET of the next job that P_i is executing (τ_{next}), exceeds r_e (Lines 24-25). Otherwise, it means that the node has spent more than WCET of jobs to finish them or has been idle while it was lagging behind other nodes. For the same reason, if adding C_{next} to the completion time of $prog_i$ on node P_i (i.e. toc_i), it must be bigger than r_e (Line 25). We determine $prog_{br}$ as the maximum between the back-runner's current progress and $prog_{br}^{prev}$ (Line 26). Similarly, toc_{br} is set to the maximum between the time when the back-runner finished its last job and toc_{br}^{prev} (Line 27). The reason behind using these maximum values is that under normal conditions, nodes always advance in progress (progress always increases), and finish their next job at a later time than the previous one. If a back-runner does not meet the conditions in Line 25, it will be considered faulty, and the next back-runner will be evaluated until the non-faulty back-runner is found (Line 29).

Finally, the algorithm updates ipo^{prev}, $prog_{br}^{prev}$ and toc_{br}^{prev} and adjust r_i^{prev} for each $\tau_i \in RB_k$ (Lines 30). It lifts the lock limit on the progress of the node P_k and finishes its execution (Line 31).

Scheduling Algorithm. Algorithm 3 illustrates our scheduling algorithm. This algorithm executes when a job finishes its execution or when the node is idle and a newly released job is inserted in the ready queue. If the progress is currently locked, the algorithm cannot advance it (Line 1); therefore, the algorithm exits and will restart when the lock limit is lifted (Line 2). Otherwise, if the algorithm is called after finishing the execution of a job, the algorithm increases the progress index and changes run_k to zero, indicating that no job is currently being executed (Lines 3-4). To execute the next job τ_a (Line 5), if P_k is already lagging behind the last known insertion point offset (Line 6), P_k executes τ_a directly (Line 7), as the execution order of jobs before ipo^{prev} is already fixed. Otherwise, the algorithm checks whether executing the next job in the ready queue would potentially violate the slack of higher priority imminent tasks on the back-runner. The algorithm constructs the set VS_a

Algorithm 3: Scheduling algorithm

```
Event     : on scheduling event
Variables : $ipo^{prev}$, $prog_{br}^{prev}$, $toc_{br}^{prev}$, $LockProg$
1  if $prog_k \geq LockProg$ // $prog_k$ is being communicated
2    exit()
3  if $run_k = 1$
4    $prog_k = prog_k + 1$; $run_k = 0$
5  $\tau_a = q_k(prog_k + 1)$
6  if $ipo^{prev} > prog_k$ // $P_k$ is not the front-runner
7    execute($\tau_a$); $run_k = 1$
8  else
9    $VS_a = \{\tau_h \in hp_a \mid t_w(prog_k) + C_a > \rho_h(t) + slack_h\}$
     // set of imminent tasks that may face priority inversion
     // exceeding their slacks if back-runner executes $\tau_a$
10   if $VS_a \neq \emptyset$
11     $IdleUntil(\max_{\tau_i \in VS_a} (t_w(prog_k) + C_a - slack_i))$
12   else
13     execute($\tau_a$); $run_k = 1$
```

(Line 9) including all imminent tasks whose earliest release plus their slack is less than $t_w(prog_k) + C_a$. VS_a includes all tasks that might miss their deadlines on the back-runner if they are executed after τ_a. If VS_a is not empty, the node remains idle until a job is released or the earliest release time of each imminent task $\tau_i \in VS_a$ is delayed enough such that $t_w(prog_k) + C_a \leq \rho_i(t) + slack_i$ (Line 11). Otherwise, the algorithm executes τ_a (Line 13).

Guaranteeing total order. All nodes consider the same release time for each release event and insert the same corresponding tasks into their ready queues. Given the same values of ipo^{prev}, $prog_{br}^{prev}$ and toc_{br}^{prev} across the nodes and the fact that they all receive the same information via the reliable broadcast channel, they can run the same protocol with the same data. Therefore, they are always able to identify the same non-faulty front-runner and back-runner, and insert the released tasks in the same position in their queues. They also update ipo^{prev}, $prog_{br}^{prev}$ and toc_{br}^{prev} to the same values. Even if a malicious node pretends to be a healthy front-runner or back-runner to influence these values, all nodes will still receive its progress information and determine the same values for these variables, thus ensuring the total order.

Ensuring Deadlines. The induced priority inversion by our method and the idle time due to locking the progress in the release algorithm or enforced by the scheduling algorithm do not cause deadline misses, provided the task set is schedulable by a non-preemptive scheduling method and the timeout does not exceed the WCET of the tasks (discussed in Section 7). Also, faulty nodes cannot exploit our method to cause healthy nodes to miss deadlines.

The priority inversion caused by inserting a released task after jobs executed by the non-faulty front-runner on lagging nodes is always constrained by the slack of the tasks. A healthy front-runner compares the t_w of its progress plus the WCET of each job that it executes with the earliest release plus the slack of imminent tasks to ensure that an imminent task, when released, will not experience priority inversion exceeding its slack. Additionally, since no lower-priority imminent task is inserted before an imminent task τ_i after its release, τ_i will never miss its deadline.

During communication, a node only remains idle when there is dynamic slack as large as *timeout* available due to executing jobs earlier than their WCETs. When there is insufficient dynamic slack, a node locks its progress at $prog_k + 1$ rather than $prog_k$, allowing the next job to execute. The completion of communication in parallel with task execution is guaranteed because the *timeout* for communication is less than the WCET of the smallest job. This ensures that the communication never causes a node to complete a job in the queue later than the t_w of its corresponding progress.

The scheduling algorithm only forces a node to idle if it executes jobs earlier than their WCETs. In the worst case, if P_k requires the WCET of jobs to execute them, it finishes each job at a time (t) which is equal to the WCP of its corresponding progress. At the time t, for an imminent task τ_i, $\rho_i(t) \geq t$, and the schedulability test requires the slack of each task to be larger than the WCET of any lower priority task. Therefore, adding the WCET of the next job in the ready queue (τ_a) to $t_w(prog_k)$ never exceeds $\rho_i(t) + slack_i$ for each imminent task τ_i. Thus, a lagging node never idles when jobs are in the ready queue and dynamic slacks are insufficient.

Faulty nodes may try to influence the insertion point of tasks on healthy nodes via Algorithm 2 by sending incorrect or missing progress data. However, the algorithm includes countermeasures to ensure task deadlines despite such attacks. A faulty node might pose as a front-runner to manipulate *ipo*. Our method prevents priority inversion attacks, ensuring that the priority inversion of the progress of the node determining *ipo* never exceeds any task's slack (Line 14); thus, deadlines are met on all nodes. However, a malicious node might behave as a healthy front-runner to determine ipo^{prev}, and then withhold or send reduced progress data in the next release. By doing so, it might try to make other nodes insert a lower-priority task ahead of a higher-priority one. Nevertheless, the algorithm ensures the insertion point offset continuously advances (Lines 8-9), preventing such attacks. Similarly, a faulty node pretending to be a back-runner may try to influence $prog_{br}$ or toc_{br} to cause deadline misses. However, the release algorithm validates the back-runner's progress and completion time (toc) based on the WCET of tasks (Line 25), and any progress data outside the defined bounds are ignored, ensuring safe $prog_{br}$ and toc_{br}. Inconsistent data, such as reporting smaller progress while $prog_{br}^{prev}$ holds a larger value, is also mitigated by the algorithm, by ensuring that $prog_{br}$ and toc_{br} are always at least as large as $prog_{br}^{prev}$ and toc_{br}^{prev} (Lines 26-27).

7 Schedulability Requirements

Our approach works for both fixed or dynamic priority non-preemptive scheduling algorithms. However, it uses the slack of tasks, which is dependent on the scheduling algorithm. Therefore, in the following, we recap the slack computations for NP-RM and NP-EDF while taking into account the overhead of the release algorithm.

Rate Monotonic. We compute the slack of tasks for fixed-priority scheduling algorithms using the method described by Yao et al. [60], and include overhead using Phan et al.'s method [49]. We first calculate the request bound function of each task τ_i which, for an interval of l, is the maximum cumulative execution time request that jobs of τ_i can generate in that interval. It is defined as [60]:

$$rbf(\tau_i, l) = \left\lceil \frac{l}{T_i} \right\rceil C_i \tag{2}$$

To include the release overhead in the analysis, we calculate the resources required by the algorithm to release jobs of τ_i within interval l as follows [49]:

$$rbf^{ISR}(\tau_i, l) = \left\lceil \frac{l}{T_i} \right\rceil \Delta^{rel} \tag{3}$$

Where Δ^{rel} is the maximum overhead to release a job. In this paper, we account only for the computational overhead of our release algorithm in Δ^{rel}. We assume that other overheads, such as interruption and scheduling, are included in the task WCET, and that message transmission occurs in parallel with task execution or uses dynamic slack available from early task completion.

Considering the overhead, the supply bound function remained for executing tasks is calculated as [49]:

$$sbf^{rem}(l) = max_{0 \leq l' \leq l}(l' - \sum_{\tau_i \in \mathbb{T}} rbf^{ISR}(\tau_i, l')) \tag{4}$$

In a fixed priority scheduling, at the critical instant [40] where all the tasks are released simultaneously, a job will be executed after all higher prioritized jobs. In a non-preemptive setting, the execution of jobs can also be blocked by the current non-preemptively executing job, which may have a lower priority. Let $slack_i$ be the maximum time that the jobs of a task τ_i can be blocked by lower-priority jobs while still meeting their deadlines. Therefore, the $slack_i$ can be calculated as [49, 60]:

$$slack_i = \max_{l \in S_i}(sbf^{rem}(l) - \sum_{\tau_j \in hep_i} rbf(\tau_j, l)) \tag{5}$$

Here, hep_i represents tasks with higher or equal priority to τ_i (including τ_i), and S_i denotes the set of points where a discontinuity occurs in the summation in Equation 5. S_i is defined as [60]:

$$S_i = \{\forall l \in [C_i, D_i] \mid l = kT_j, k \in \mathbb{N} \text{ } and \text{ } \forall \tau_j, D_j \leq D_i\} \tag{6}$$

A task-set is schedulable by NP-RM if the non-preemptive execution of tasks, which could be as large as their WCETs, does not exceed the slack of higher prioritized tasks. That is:

$$\forall \tau_i \in \mathbb{T}, \max_{\tau_j \in lp_i}(C_j) \leq slack_i \tag{7}$$

Earliest Deadline First. Our method evaluates job priorities only at run-time. Once a job is released, its absolute deadline and hence the priority for EDF is fixed. Similarly, for an imminent task τ_i, we can at time t compute its earliest possible absolute deadline based on its earliest possible release time $\rho_i(t)$ plus its relative deadline D_i. We then derive its priority based on this earliest absolute deadline.

We leverage the results of Bertogna et al. [8] and Phan et al. [49] to compute the slack of a task τ_i based on its demand bound function dbf_i. The demand bound function of a task τ_i denotes the maximum amount of time that τ_i requires in a given interval of length l to finish all jobs that it might release during that interval [6]:

$$dbf(\tau_i, l) = (\left\lfloor \frac{l - D_i}{T_i} \right\rfloor + 1).C_i \tag{8}$$

A task set is schedulable using preemptive EDF, if [6, 49]:

$$\forall l \in X_i : (sbf^{rem}(l) - \sum_{\tau_i \in \mathbb{T}} dbf(\tau_i, l)) > 0 \tag{9}$$

where the set X_i contains all absolute deadlines before the hyperperiod HP (i.e., the least common multiple of all minimum inter-release times T_i) [6, 8]:

$$X_i = \{l \in [0, HP] \mid l = kT_j + D_j, k \in \mathbb{N} \text{ and } \tau_j \in \mathbb{T}\} \tag{10}$$

In preemptive EDF, a job can only be delayed by the jobs with earlier absolute deadlines, which have higher priorities; therefore, a job never experiences priority inversion. However, in non-preemptive EDF, the execution of a job can also be blocked by non-preemptive execution of lower priority jobs. A job of a task can be subject to priority inversion only from tasks with larger relative deadlines. Therefore, we calculate the slack of each task as the maximum time that we can delay its jobs by non-preemptively executing jobs that have larger relative deadlines. The task with the largest relative deadline (D_{max}) cannot experience priority inversion. Considering the overhead of the release algorithm, the slack of a task τ_i, where $D_i \neq D_{max}$, is calculated as [8, 49]:

$$slack_i = \min_{l \in X_i \cap I} (sbf^{rem}(l) - \sum_{\tau_j \in \mathbb{T}} dbf(\tau_j, l)). \tag{11}$$

where $I = [D_i, \min_{\{\tau_j \neq \tau_i | D_i < D_j\}} (D_j))$ is the interval that is bounded from below by D_i and from above by the smallest relative deadline among all tasks that have larger relative deadlines than D_i. A task set $\mathbb{T}$ is schedulable by NP-EDF (and our method) if Equation 9 is true for each time interval $l \in X_i$ [6] and if for each task τ_i with $D_i \neq D_{max}$, $slack_i$ is greater than (or equal to) the WCET C_j of other tasks τ_j with larger or equal relative deadlines. That is [8]:

$$\forall \tau_i \in \mathbb{T} | D_i \neq D_{max}, \max_{\{\tau_j \in \mathbb{T}, j \neq i | D_i \leq D_j\}} (C_j) \leq slack_i \tag{12}$$

Our method, at run-time, allows nodes to use the entire slack of a high-priority imminent task to make independent progress, but the priority inversion never exceeds the slack of the task.

Communication bound. It is important to ensure that the *timeout* (and thus the maximum communication delay) does not exceed the WCET of the smallest task. This is because, once a task releases at r_e and the node initiates communication, the node will either remain idle until $r_e + timeout$ after finishing the ongoing task or execute the next job in the queue. The former occurs when there is enough dynamic slack available on the node until $r_e + timeout$. The latter occurs when a node P_b finishes the execution of jobs close to their WCETs. If the *timeout* is bounded by the WCET of tasks, it guarantees that the time by which P_b finishes a job at progress $prog_b$ never exceeds $t_w(\text{prog}_b)$ (as explained in Section 6). Additionally, it ensures that any priority inversion caused by P_b executing a lower-priority job until $r_e + timeout$ remains within the slack bounds of the released task. This is because the slack of tasks is always greater than the WCET of lower-priority tasks.

In real-world systems, communication *timeout* and the range of the WCET of tasks depends on the system specifications. For example, when using communication channels such as high-speed variants of the CAN Bus, CAN FD or CAN XL, the communication delay remains within the range of microseconds, while the WCETs of application level tasks are typically much larger.

8 Evaluation

We implemented our algorithms in C++ and conducted a simulation-based evaluation to assess the performance of our approach. We

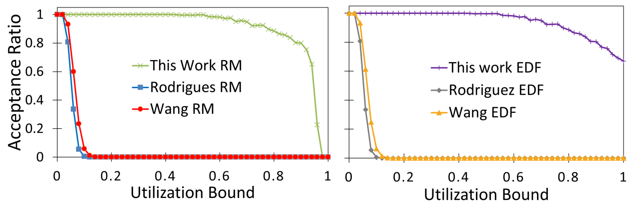


Figure 3: Fraction of task sets that remain schedulable under Rodrigues, Wang, and our approach for RM and EDF.

compared our approach in terms of acceptance ratio and response time against the algorithms proposed by Rodrigues et al.[52], Wang et al.[57], and the *Simple* method. The Simple method is the non-work-conserving algorithm presented in Section 1 which requires nodes to idle until the WCET of a job, even if this job is completed earlier. We also measured the run-time overhead of our algorithms on a Raspberry Pi 4B, running at 1.5 GHz.

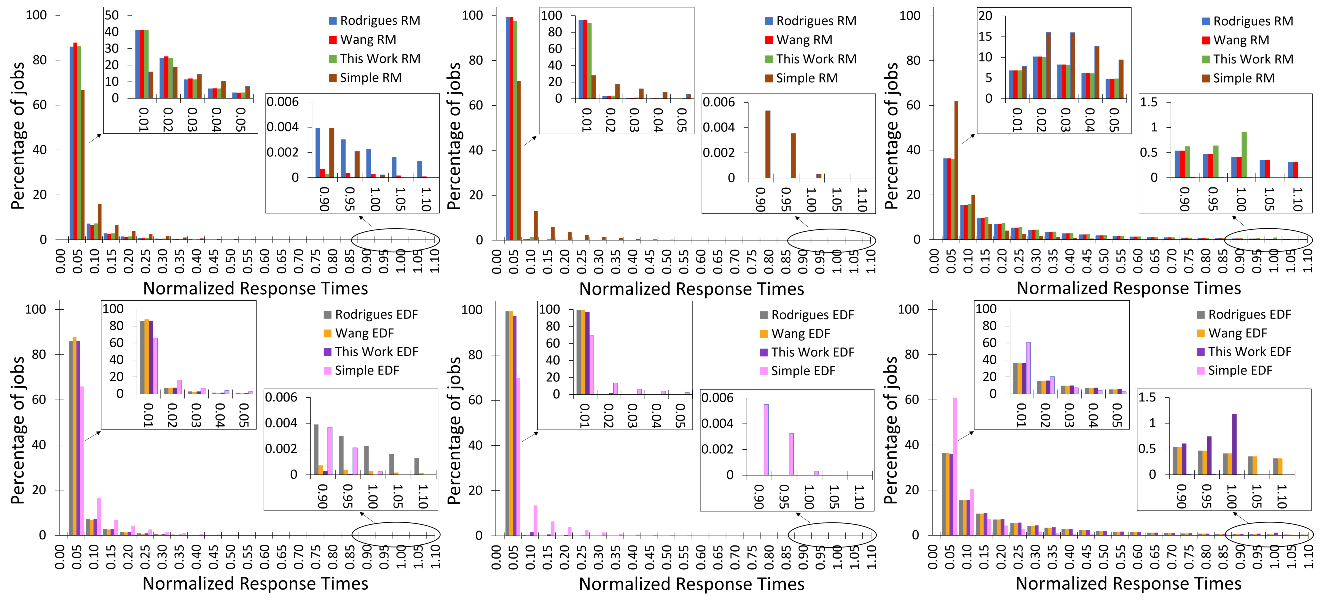
Experimental setup. We derive task sets from TACLeBench [17], following the approach that is used in [41]. That is, we measured the execution time of all TACLeBench tasks on the Raspberry Pi, running Linux, and used the maximum observed execution time of tasks (after eliminating outliers) as their WCETs. We set the best case execution time (BCET) of tasks to 20% of their WCETs [5, 58] and consider only implicit deadlines (i.e., $D_i = T_i$) for simplicity. Note that, our approach works equally well for tasks with $D_i \leq T_i$.

We generate task sets composed of 100 tasks, by using the Dirichlet-Rescale algorithm [22] to obtain 100 uniformly distributed random utilizations ($u_1, \ldots, u_{100}$), which sum up to the targeted utilization bound U. From these, we calculate the minimum inter-release time of each task as $T_i = C_i/u_i$ for a randomly selected task out of the 15 tasks from TACLeBench that have a WCET between 1ms and 100ms. This selection criterion helps maintain uniformity by avoiding a mix of tasks with significantly different WCETs.

In these experiments, we assume that the transmission time of messages over the network is bounded by a timeout of $10\mu s$.

Acceptance Ratio. Our approach requires the slack of tasks to be at least as large as the WCET of all lower-prioritized tasks for NP-RM and all tasks with larger relative deadlines for NP-EDF. This requirement, which is the same criterion for a standard single processor NP-RM and NP-EDF scheduling (hence, the Simple method), limits the schedulable task sets. Figure 3 shows the schedulable utilization for our approach as well as Rodrigues' and Wang's approaches for different utilization bounds. For each bound, we generated 1000 task sets and reported the schedulable fraction, indicating the proportion of task sets where tasks can meet their deadlines even if f out of $2f + 1$ nodes are faulty. Since the WCET of each task is the maximum of the WCET of its replicas, f can be chosen arbitrarily without affecting the results.

Under Rodrigues' protocol, one malicious node is enough to perform the priority inversion attack. In the worst case, this node can report having executed one job of each lower-priority task when a job of task τ_i is released (similarly, one job of each task with a larger relative deadline for NP-EDF). This assumes that job instances of a task are released at least the minimal inter-release time apart. Task sets with sufficient slack to accommodate the priority inversion from such attacks can still meet their deadlines, which we show as Rodrigues' acceptance ratio for comparison. Given the slacks



(a) normal scenario (all nodes) **(b) extreme scenario (front-runner)** **(c) extreme scenario (back-runner)**

Figure 4: Impact of different methods on the response time of the tasks with total utilization of 0.9 normalized by their deadlines. The figure shows results for all nodes in the normal scenario (left), the healthy front-runner node (middle), and the back-runner node (right) in the extreme scenario. Response times for RM scheduling are presented at the top and EDF at the bottom.

calculated differently for RM and EDF based on Section 7, and considering implicit deadlines, the criteria to guarantee the deadlines of tasks in Rodrigues is the same for both RM and EDF, which is $\forall \tau_i \in \mathbb{T}, \sum_{\tau_j \in \mathbb{T}, j \neq i | D_j \geq D_i} C_j \leq slack_i$.

A similar attack applies to Wang's method if f malicious nodes out of $2f + 1$ nodes report early execution of jobs with their BCETs and a healthy node executes jobs earlier than other nodes (e.g., with their BCETs). Priority inversion is injected by requiring the other nodes to catch up. Again this can be compensated by large enough slacks. Considering implicit deadlines, a task set is schedulable by Wang with RM and EDF (even when the non-faulty front-runner execute jobs with their BCETs), if $\forall \tau_i \in \mathbb{T}, C_k + \sum_{\tau_j \in \mathbb{T} - \{\tau_k, \tau_i\} | D_j \geq D_i} (C_j - BCET_j) \leq slack_i$ where $C_k = max_{\tau_j \in \mathbb{T}, j \neq i | D_j \geq D_i} C_j$.

As shown in Figure 3, our method demonstrates a considerably higher acceptance ratio than Rodrigues and Wang methods. Even with a total utilization of 0.9, for RM and EDF, 80% of the task sets have been schedulable using our method. However, in every generated task set with a total utilization as low as 0.12 for Rodrigues and 0.16 for Wang (for RM and EDF), these methods might miss the deadline of tasks under priority inversion injection attacks.

We repeated the experiments with tasks having BCETs at 50% of their WCETs. In this scenario, the acceptance ratios of our method and Rodrigues remained unchanged. However, the Wang method, which depends on the difference between WCET and BCET, showed a slight improvement but still had an acceptance ratio of 0 for total utilizations of 0.24 and above.

Response Time. Although real-time systems are primarily concerned with the worst-case execution behavior of tasks, there are several secondary concerns where the actual response time of jobs plays a crucial role (e.g., power management, background load, mixed-criticality, etc.). In the following, we show the response time distribution for a system with 5 nodes (i.e., $f = 2$) and 100 task sets, with a utilization bound of 0.9 and 100 tasks each, generated as described in the experimental setup. Each task set is schedulable according to the criteria mentioned in Equations 7 and 12.

We evaluate the execution of 100,000 jobs in two scenarios. In the first, *normal scenario*, actual execution times are randomly drawn from the job's $[BCET, WCET]$ interval. In the second scenario, we simulate an *extreme case* where f nodes are malicious, one healthy node executes jobs with their BCETs, while another healthy node requires jobs' full WCETs to finish them. We assume the malicious nodes falsely report job execution times as their BCETs. In these experiments, the release times of sporadic tasks τ_i are chosen randomly, ensuring that the interval between consecutive releases are bigger than T_i, but smaller than $2T_i$, thereby, no additional jobs of the same task can be added between two consecutive jobs.

Figure 4a shows histograms of normalized response times of jobs on all nodes scheduled by NP-RM and NP-EDF in the normal scenario. Figure 4b, and 4c respectively show the normalized response times on the healthy front-runner and back-runner in the extreme scenario (for NP-RM and NP-EDF). Normalized response times of jobs are their actual response times relative to their release, divided by their relative deadlines. We plot the fraction of jobs whose normalized response time falls into the corresponding bin of size 0.05. We also zoom into the $(0, 0.05]$ bin and the range above 0.90.

As can be seen, there are jobs with normalized response times bigger than 1 for Rodrigues' and Wang's methods. That is, jobs exist that these methods complete only after their deadline. For NP-RM in the normal scenario, 0.0084% of the jobs missed their deadlines when using Rodrigues' method and 0.0005% when using

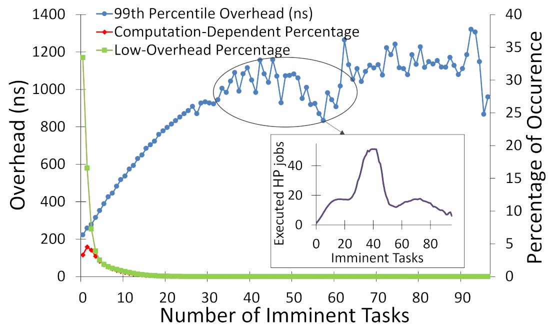
Wang's method. In the extreme scenario and with NP-RM, on the back-runner, the response time of 2.896% of the jobs exceeded their deadlines for both Rodrigues and Wang. In this scenario, both Rodrigues' and Wang's methods perform similarly, as the back-runner node follows the execution order of a node executing jobs with their BCETs. Our approach guarantees the completion of all jobs before their deadlines in both scenarios.

Our method outperforms the Simple method in terms of average response time. For instance, under the normal scenario using NP-RM, our method resulted in 65.177% of jobs having a normalized response time of less than 0.02, whereas only 34.629% of jobs achieved this with the Simple method. In the extreme scenario (Figures 4b and 4c), our method significantly improves job response times on the front-runner compared to the Simple method but increases response times on the back-runner. This occurs because the back-runner inserts the released jobs after all jobs the healthy front-runner executes. This blocks the execution of recently released jobs on the back-runner by first executing multiple lower-priority jobs to match the execution order of the front-runner.

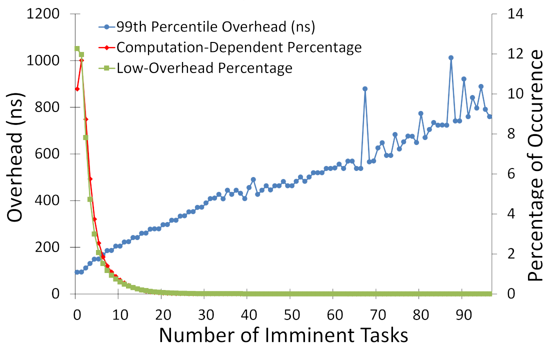
We repeated these experiments considering BCET of 50% compared to the WCET of tasks. In these measurements, the overall trend (not shown) remained consistent with Figure 4, although the response times of jobs scheduled using different methods were slightly more similar to each other. In the normal scenario, our method achieved a normalized response time under 0.02 for 48.46% of jobs, compared to 29.79% for the Simple method. Rodrigues and Wang had deadline miss rates of 0.0074% and 0.0003%, respectively, in the normal scenario, and missed 2.435% of deadlines in the extreme scenario. As expected, our method did not miss any deadlines.

Overhead. To evaluate the overhead of our method, we ran our algorithms on a Linux-based quad-core Raspberry Pi 4B (8GB RAM, 1.5 GHz) and monitored their execution time. In these experiments, we focus on the computational overhead of our implemented release and scheduling algorithms, which are optimized to check the slack of each imminent task at most once. The overhead of the release algorithm includes calculating ipo, $prog_{br}$, and toc_{br} at each release, while the scheduling algorithm's overhead involves deciding whether to execute the next job or remain idle. The overhead of task interruptions, context switching, and enqueuing tasks into the ready queue, which is inherent to all methods, is not included in these measurements and is assumed to be covered by the WCET of the tasks. Additionally, communication overhead is not included, assuming it is managed by a dedicated component integrated into the processor. We evaluate the overhead of our method for schedulable task sets of 100 tasks, generated according to the experimental settings, with a total utilization of 0.9. We conducted the experiments for 500 task sets, executing 100,000 jobs per task set.

The runtime overhead of the computations in both the release and scheduling algorithms primarily depends on the number of imminent tasks that our algorithms examine their slacks. However, there are situations where the release algorithm efficiently locates the insertion point and the scheduling algorithm can decide about the execution of the next job without checking these slacks. In these cases, which we consider low-overhead cases, the overhead is typically less than 0.1 microseconds. Figure 5 illustrates the frequency of low-overhead situations compared to situations with higher overheads. It also presents the computational run-time



(a) Release



(b) Scheduling

Figure 5: Frequency and overhead of slack checking in release and scheduling algorithms.

overhead of each algorithm based on the number of imminent tasks they checked. The overhead values depicted in the graph represent the 99th percentile of all algorithm executions for the specified number of imminent tasks. This percentile is used instead of the maximum to filter out measurement-related outliers.

For the release algorithm (Figure 5a), the overhead consistently remains below or around 1.3 microseconds, and in 99.91% of cases, it is less than 0.8 microseconds. In the release algorithm, the relationship between overhead and the number of imminent tasks is not linear. This is because the algorithm also traverses the jobs in the ready queue to identify the non-faulty front-runner. This aspect correlates directly with the number of imminent tasks whose slacks are checked which is detailed in a smaller graph within Figure 5a. The overhead for checking the slack of imminent tasks in the scheduling algorithm (Figure 5b) is linear with the number of imminent tasks and its observed maximum is $1\mu s$. In 99.94% of the times, our scheduling algorithm finishes with an overhead of less than $0.4\mu s$.

9 Conclusions

In this work, we have described a novel time-domain attack to priority based total order protocols in real-time systems by which faulty nodes can inject priority inversion into their peers, which are following an agreed upon execution order. Analyzing two protocols by Rodrigues et al. and Wang et al., we found both to be vulnerable to these attacks. We have designed, implemented and evaluated a method that coordinates the replicated execution of sporadic tasks in event-triggered real-time systems and that is robust against priority-inversion injection attacks. Using the slack of imminent tasks, our approach allows nodes to progress independently, requires no roll-backs and improves the response time of jobs compared to the Simple solution that always idles until the worst-case execution time of jobs after finishing their execution.

Directions for future works include extending our work to less tightly coupled nodes with coarsely synchronized clocks, and larger communication delays. Using milestones in the tasks, we will also extend our method to guarantee the same execution sequence of replicated tasks among nodes in preemptive task models.

Acknowledgments

This research was funded in whole or in part by the Luxembourg National Research Fund (FNR), grant reference C19/IS/13691843/-ByzRT. For the purpose of open access, and in fulfillment of the obligations arising from the grant agreement, the author has applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.

References

[1] Hakan Aydin Abhishek Roy and Dakai Zhu. 2021. Energy-aware primary/backup scheduling of periodic real-time tasks on heterogeneous multicore systems. *Sustainable Computing: Informatics and Systems* 29 (2021), 100474. https://doi.org/10.1016/j.suscom.2020.100474

[2] Mani Amoozadeh, Arun Raghuramu, Chen-nee Chuah, Dipak Ghosal, H. Michael Zhang, Jeff Rowe, and Karl Levitt. 2015. Security vulnerabilities of connected vehicle streams and their impact on cooperative driving. *IEEE Communications Magazine* 53, 6 (2015), 126–132. https://doi.org/10.1109/MCOM.2015.7120028

[3] Tom Roeder an Fred B. Schneider. 2010. Proactive Obfuscation. *ACM Transactions on Computer Systems (TOCS)* 28, 2 (July 2010).

[4] Alaa Askkar. 2011. PA Telecommunications minister: Palestinian Internet Under Hacking Attacks. IMENC avail at. http://www.imemc.org/article/62409.

[5] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. 2001. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*. 95–105. https://doi.org/10.1109/REAL.2001.990600

[6] Sanjoy Baruah. 2005. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. 137–144. https://doi.org/10.1109/ECRTS.2005.32

[7] Guillem Bernat, Jose Miro-Julia, Julian Proenza, et al. 1997. Fixed Priority Schedulability Analysis of a Distributed Real-Time Fault Tolerant Architecture.. In *PDPTA*. Citeseer, 479–487.

[8] Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni, Gang Yao, Francesco Esposito, and Marco Caccamo. 2010. Preemption Points Placement for Sporadic Task Sets. In *2010 22nd Euromicro Conference on Real-Time Systems*. 251–260. https://doi.org/10.1109/ECRTS.2010.9

[9] N.L. Binkert, L.R. Hsu, A.G. Saidi, R.G. Dreslinski, A.L. Schultz, and S.K. Reinhardt. 2005. Performance analysis of system overheads in TCP/IP workloads. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 218–228. https://doi.org/10.1109/PACT.2005.35

[10] Giorgio C. Buttazzo, Marko Bertogna, and Gang Yao. 2013. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *IEEE Transactions on Industrial Informatics* 9, 1 (2013), 3–15. https://doi.org/10.1109/TII.2012.2188805

[11] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (nov 2002), 398–461. https://doi.org/10.1145/571637.571640

[12] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. 2011. Comprehensive Experimental Analyses of Automotive Attack Surfaces.. In *USENIX Security Symposium*. San Francisco.

[13] US Federal Energy Regulatory Commission. 2016. Reliability Standards for Physical Security Measures. RD14-6-000.

[14] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. 2013. BFT-TO: Intrusion Tolerance with Less Replicas. *Comput. J.* 56, 6 (2013), 693–715. https://doi.org/10.1093/comjnl/bxs148

[15] Marco Dürr, Georg Von Der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. 2019. End-to-End Timing Analysis of Sporadic Cause-Effect Chains in Distributed Systems. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 58 (oct 2019), 24 pages. https://doi.org/10.1145/3358181

[16] Reinhard Exel, Thomas Bigler, and Thilo Sauter. 2014. Asymmetry Mitigation in IEEE 802.3 Ethernet for High-Accuracy Clock Synchronization. *IEEE Transactions on Instrumentation and Measurement* 63, 3 (2014), 729–736. https://doi.org/10.1109/TIM.2013.2280489

[17] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis*. Toulouse, France. https://doi.org/10.4230/OASIcs.WCET.2016.2

[18] Pietro Fara, Gabriele Serra, Alessandro Biondi, and Ciro Donnarumma. 2021. Scheduling Replica Voting in Fixed-Priority Real-Time Systems. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 196)*, Björn B. Brandenburg (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13:1–13:21. https://doi.org/10.4230/LIPIcs.ECRTS.2021.13

[19] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. 2009. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *IEEE Real-Time Systems Symposium: 30/11/2009-03/12/2009*. IEEE Communications Society.

[20] Joachim Fellmuth, Paula Herber, Tobias Pfeffer, and Sabine Glesner. 2017. Securing Real-Time Cyber-Physical Systems Using WCET-Aware Artificial Diversity. 454–461. https://doi.org/10.1109/DASC-PICom-DataCom-CyberSciTec.2017.88

[21] Laurent George, Nicolas Rivierre, and Marco Spuri. 1996. *Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling*. Research Report RR-2966. INRIA. https://hal.inria.fr/inria-00073732 Projet REFLECS.

[22] David Griffin, Iain Bate, and Robert I. Davis. 2020. Generating Utilization Vectors for the Systematic Evaluation of Schedulability Tests. In *2020 IEEE Real-Time Systems Symposium (RTSS)*. 76–88. https://doi.org/10.1109/RTSS49844.2020.00018

[23] Arpan Gujarati, Sergey Bozhko, and Björn B. Brandenburg. 2020. Real-Time Replica Consistency over Ethernet with Reliability Bounds. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 376–389. https://doi.org/10.1109/RTAS48715.2020.00012

[24] Mario Günzel, Harun Teper, Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. 2023. On the Equivalence of Maximum Reaction Time and Maximum Data Age for Cause-Effect Chains. In *35th Euromicro Conference on Real-Time Systems (ECRTS 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 262)*, Alessandro V. Papadopoulos (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:22. https://doi.org/10.4230/LIPIcs.ECRTS.2023.10

[25] Zhishan Guo, Sudharsan Vaidhun, Abdullah Al Arafat, Nan Guan, and Kecheng Yang. 2023. Stealing Static Slack Via WCRT and Sporadic P-Servers in Deadline-Driven Scheduling. In *2023 IEEE Real-Time Systems Symposium (RTSS)*. 40–52. https://doi.org/10.1109/RTSS59052.2023.00014

[26] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B. Bobba. 2018. A design-space exploration for allocating security tasks in multicore real-time systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 225–230. https://doi.org/10.23919/DATE.2018.8342007

[27] K. Jeffay, D.F. Stanat, and C.U. Martel. 1991. On non-preemptive scheduling of period and sporadic tasks. In *[1991] Proceedings Twelfth Real-Time Systems Symposium*. 129–139. https://doi.org/10.1109/REAL.1991.160366

[28] Gregg Keizer. 2010. Is Stuxnet the 'best' malware ever? avail at. http://www.infoworld.com/article/2626009/malware/is-stuxnet-the--best--malware-ever-.html.

[29] Uğur Keskin, Reinder J. Bril, and Johan J. Lukkien. 2010. Exact response-time analysis for fixed-priority preemption-threshold scheduling. In *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation*. 1–4. https://doi.org/10.1109/ETFA.2010.5640984

[30] J. Kim, G. Park, H. Shim, and Y. Eun. 2016. Zero-stealthy attack for sampled data control systems: The case of faster actuation than sensing. In *IEEE Conference on Decision and Control (CDC)*. 5956––5961.

[31] K.H. Kim, Jing Qian, Zhen Zhang, Qian Zhou, Kyung-Deok Moon, Jun-Hee Park, Kwang-Roh Park, and Doo-Hyun Kim. 2010. A Scheme for Reliable Real-Time Messaging with Bounded Delays. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 18–27. https://doi.org/10.1109/ISORC.2010.45

[32] Leonie Köhler, Phil Hertha, Matthias Beckert, Alex Bendrick, and Rolf Ernst. 2023. Robust Cause-Effect Chains with Bounded Execution Time and System-Level Logical Execution Time. *ACM Trans. Embed. Comput. Syst.* 22, 3, Article 50 (apr 2023), 28 pages. https://doi.org/10.1145/3573388

[33] H. Kopetz and G. Bauer. 2003. The time-triggered architecture. *Proc. IEEE* 91, 1 (2003), 112–126. https://doi.org/10.1109/JPROC.2002.805821

[34] H. Kopetz and G. Grunsteidl. 1993. TTP - A time-triggered protocol for fault-tolerant real-time systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. 524–533. https://doi.org/10.1109/FTCS.1993.627355

[35] D. Kozhaya, J. Decouchant, and P. Esteves-Verissimo. 2019. RT-ByzCast: Byzantine-Resilient Real-Time Reliable Broadcast. *IEEE Trans. Comput.* 68, 03 (mar 2019), 440–454. https://doi.org/10.1109/TC.2018.2871443

[36] David Kozhaya, Jérémie Decouchant, Vincent Rahli, and Paulo Esteves-Verissimo. 2021. PISTIS: An Event-Triggered Real-Time Byzantine-Resilient Protocol Suite.

IEEE Transactions on Parallel and Distributed Systems 32, 9 (2021), 2277–2290. https://doi.org/10.1109/TPDS.2021.3056718

[37] Kristin Krüger, Nils Vreman, Richard Pates, Martina Maggio, Marcus Völp, and Gerhard Fohler. 2021. Randomization as Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Real-Time Systems with Task Replication. 7 (Aug. 2021), 01:1–01:29. https://doi.org/10.4230/LITES.7.1.1

[38] Robert M. Lee, Michael J. Assante, and Tim Conway. 2016. Analysis of the Cyber Attack on the Ukrainian Power Grid. E-ISAC avail at. https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf.

[39] Haoran Li, Chenyang Lu, and Christopher D. Gill. 2021. RT-ZooKeeper: Taming the Recovery Latency of a Coordination Service. *ACM Trans. Embed. Comput. Syst.* 20, 5s, Article 103 (Sept. 2021), 22 pages. https://doi.org/10.1145/3477034

[40] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (jan 1973), 46–61. https://doi.org/10.1145/321738.321743

[41] Filip Marković, Jan Carlson, Sebastian Altmeyer, and Radu Dobrin. 2020. Improving the Accuracy of Cache-Aware Response Time Analysis Using Preemption Partitioning. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 165)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:23. https://doi.org/10.4230/LIPIcs.ECRTS.2020.5

[42] Odorico Machado Mendizabal, Fernando Luís Dotti, and Fernando Pedone. 2017. High Performance Recovery for Parallel State Machine Replication. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 34–44. https://doi.org/10.1109/ICDCS.2017.193

[43] Jeanne Meserve. 2007. Mouse click could plunge city into darkness, experts say. http://edition.cnn.com/2007/US/09/27/power.at.risk/index.html. Accessed: 2017-03-12.

[44] Amin Naghavi, Sepideh Safari, and Shaahin Hessabi. 2021. Tolerating Permanent Faults with Low-Energy Overhead in Multicore Mixed-Criticality Systems. *IEEE Transactions on Emerging Topics in Computing* (2021), 1–1. https://doi.org/10.1109/TETC.2021.3059724

[45] Mitra Nasri, Thidapat Chantem, Gedare Bloom, and Ryan M. Gerdes. 2019. On the Pitfalls and Vulnerabilities of Schedule Randomization Against Schedule-Based Attacks. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 103–116. https://doi.org/10.1109/RTAS.2019.00017

[46] Risat Mahmud Pathan. 2014. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Syst.* 50 (2014), 509–547. https://doi.org/10.1016/j.suscom.2020.100474

[47] Veríssimo P.E., Neves N.F., and Correia M.P. 2003. Intrusion-Tolerant Architectures: Concepts and Design. *Architecting Dependable Systems. Lecture Notes in Computer Science* 2677 (2003). https://doi.org/10.1007/3-540-45177-3_1

[48] M. Pease, R. Shostak, and L. Lamport. 1980. Reaching Agreement in the Presence of Faults. *J. ACM* 27, 2 (April 1980), 228–234. https://doi.org/10.1145/322186.322188

[49] Linh T. X. Phan, Meng Xu, Jaewoo Lee, Insup Lee, and Oleg Sokolsky. 2013. Overhead-aware compositional analysis of real-time systems. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 237–246. https://doi.org/10.1109/RTAS.2013.6531096

[50] S. Poledna, A. Burns, A. Wellings, and P. Barrett. 2000. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Trans. Comput.* 49, 2 (2000), 100–111. https://doi.org/10.1109/12.833107

[51] R. Pucella and F. B. Schneider. 2006. Independence from obfuscation: A semantic framework for diversity.. In *19th IEEE Work. on Computer Security Foundations*. 230–241.

[52] Luís Rodrigues, Paulo Veríssimo, and Antonio Casimiro. 1995. Priority-Based Totally Ordered Multicast. In *3rd IFIP/IFAC workshop on Algorithms and Architectures for Real-Time Control (AARTC)*.

[53] Edo Roth and Andreas Haeberlen. 2021. Do Not Overpay for Fault Tolerance!. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '21)*. https://doi.org/10.1109/RTAS52030.2021.00037

[54] F Schneider. 2005. Replication management using the statemachine approach. *Mullender [7]* (2005).

[55] Brad Spengler. 2003. PaX: The Guaranteed End of Arbitrary Code Execution. grsecurity.net avail at. https://grsecurity.net/PaX-presentation.pdf.

[56] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (2013), 16–30. https://doi.org/10.1109/TC.2011.221

[57] Yun Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin. 2002. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Trans. Comput.* 51, 8 (2002), 900–915. https://doi.org/10.1109/TC.2002.1024738

[58] Yi wen Zhang and Rui feng Guo. 2013. Power-aware scheduling algorithms for sporadic tasks in real-time systems. *Journal of Systems and Software* 86, 10 (2013), 2611–2619. https://doi.org/10.1016/j.jss.2013.04.075

[59] Beyazit Yalcinkaya, Mitra Nasri, and Björn B. Brandenburg. 2019. An Exact Schedulability Test for Non-Preemptive Self-Suspending Real-Time Tasks. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, Jürgen Teich and Franco Fummi (Eds.). IEEE, 1228–1233. https://doi.org/10.23919/DATE.2019.8715111

[60] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. 2009. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 351–360.

[61] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. 2016. 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)*. 1–12. https://doi.org/10.1109/RTAS.2016.7461362

[62] Lin Zhang, Kaustubh Sridhar, Mengyu Liu, Pengyuan Lu, Xin Chen, Fanxin Kong, Oleg Sokolsky, and Insup Lee. 2023. Real-Time Data-Predictive Attack-Recovery for Complex Cyber-Physical Systems. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 209–222. https://doi.org/10.1109/RTAS58335.2023.00024

[63] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine Ordered Consensus without Byzantine Oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 633–649. https://www.usenix.org/conference/osdi20/presentation/zhang-yunhao

[64] H. Zou and F. Jahanian. 1998. Real-time primary-backup (RTPB) replication with temporal consistency guarantees. In *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No.98CB36183)*. 48–56. https://doi.org/10.1109/ICDCS.1998.679486