Blinder: Partition-Oblivious Hierarchical Scheduling

Man-Ki Yoon, Mengqi Liu, Hao Chen, Jung-Eun Kim, Zhong Shao
Yale University

Abstract

Hierarchical scheduling enables modular reasoning about the temporal behavior of individual applications by partitioning CPU time and thus isolating potential misbehavior. However, conventional time-partitioning mechanisms fail to achieve strong temporal isolation from a security perspective; variations in the executions of partitions can be perceived by others, which enables an algorithmic covert timing-channel between partitions that are completely isolated from each other in the utilization of time. Thus, we present a run-time algorithm that makes partitions oblivious to others' varying behaviors even when an adversary has full control over their timings. It enables the use of dynamic time-partitioning mechanisms that provide improved responsiveness, while guaranteeing the algorithmic-level non-interference that static approaches would achieve. From an implementation on an open-source operating system, we evaluate the costs of the solution in terms of the responsiveness as well as scheduling overhead.

1 Introduction

With advancement in modern computing and communication technologies, there has been an increasing trend toward integrating a number of software applications into a high-performance system-on-chip to reduce communication and maintenance complexity, while allowing them to efficiently utilize common hardware resources. Such a composition requires that properties established for individual subsystems be preserved at the integrated-system level. Especially for safety-critical systems (e.g., avionics, automotive, industrial control systems), it is of utmost importance to provide strong isolation among applications that require different levels of criticality in order to limit *interference* among them: protecting high-critical applications (e.g., instrument cluster in digital cockpits [4]) from faulty behaviors of lower-critical ones (e.g., infotainment system). This is increasingly challenging as individual subsystems are becoming more complex due to advanced capabilities.

The isolation among subsystem applications is attained via a form of *resource partitioning* [33]. For example, ARINC 653 (Avionics Application Standard Software Interface) [8] standardizes time and space partitioning of applications according to their design-assurance levels. This enables the software functions to be developed and certified independently. In particular, *time* partitioning is a key ingredient for providing strong temporal-isolation of unpredictable interference from timing-sensitive applications. It is enforced also in Multiple Independent Levels of Security (MILS) systems [9] to prevent a compromised application from exhausting time resources. Operating in the form of *two-level hierarchical scheduling* architecture [6, 14], as shown in Figure 1, it provides each application with the illusion of exclusive CPU resources.

However, such a tight integration of applications poses a security challenge that could have been easily addressed in the air-gapped environment. In particular, we observe that conventional hierarchical scheduling schemes enable illegitimate *information-flow* among partitions that are supposed to be isolated from one another. As multiple threads sharing the CPU time can collaborate to influence one's execution progress and infer secret information [41, 44], time-partitions can form *algorithmic covert timing-channel* by varying their temporal behavior. In this paper, we first demonstrate such a vulnerability of hierarchical scheduling on existing real-time operating systems; a partition can infer another partition's varying execution by observing the impact that the latter's execution has on its own local schedule, even without using any time information. This algorithmic channel exists even if microarchitectural channels [15, 24] were completely closed.

Based on these observations, we develop a run-time schedule transformation algorithm that we call Blinder. It prevents partitions from distinguishing others' varying execution behavior by making each partition-local schedule *deterministic*. Although static partitioning approaches, such as TDMA (Time Division Multiple Access) [8], can achieve strong non-interference due to the fixed nature of the schedules that they generate, they are inflexible in CPU resource usage [17, 22]. On the other hand, non-static partitioning schemes, such as real-time server algorithms [7, 37, 39], are more amenable to dynamic workload and achieve better responsiveness, and thus are increasingly adopted by commercial real-time operating systems and hypervisors [3, 5]. However, such a non-determinism, or flexibility, is a double-edged sword as it is the very source of information-flow between partitions; partitions can use CPU time in a detectable pattern to send signals.

Our Blinder deters such attempts by guaranteeing that the local-execution state (i.e., partition-local schedule) changes at deterministic time points no matter how the global state

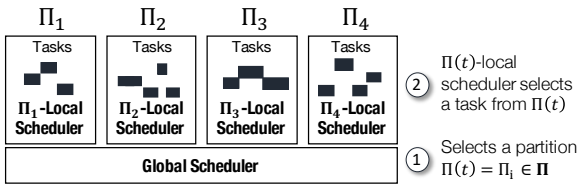


Figure 1: Hierarchical scheduling.

changes (i.e., partition-level schedule). Hence, partitions can be scheduled still in non-static ways (thus taking advantage of the flexibility in resource utilization) while achieving the strong *partition-obliviousness* property that has been possible only with static approaches. Blinder removes the algorithmic covert timing-channel in hierarchical scheduling even if an attacker is able to precisely control the timings of applications. Furthermore, it is minimally-intrusive and modular in that it does not require any modifications to the global and local scheduling policies and hence can be applied to a wide variety of existing systems that employ hierarchical scheduling.

In summary, this paper makes the following contributions:

- We demonstrate an algorithmic covert timing-channel between time-partitions through hierarchical scheduler of existing real-time operating systems.
- We introduce Blinder, a run-time schedule transformation algorithm that makes partitions oblivious to others' varying temporal behavior, and we also analyze its impact on the schedulability of real-time tasks.
- We implement Blinder on an open-source real-time operating system and evaluate its impact on the responsiveness as well as scheduling overheads.
- We demonstrate a deployment of Blinder on a prototype real-time system and discuss system design and implementation challenges and possible mitigations.

2 Preliminaries

2.1 System Model and Terminology

We consider a real-time system that hosts N applications, $\mathbf{\Pi} = \{\Pi_1, \ldots, \Pi_N\}$, sharing CPU time. Each application, or *partition*, Π_i is comprised of one or more *tasks*, i.e., $\Pi_i = \{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,m_i}\}$, where m_i is the number of tasks in Π_i. Each task is characterized by $\tau_{i,j} := (p_{i,j}, e_{i,j})$, where $p_{i,j}$ is the minimum inter-arrival time (i.e., period) and $e_{i,j}$ is the worst-case execution time.

The partitions are scheduled in a *hierarchical* manner [8, 14] as shown in Figure 1 and Algorithm 1; the *global* scheduler determines which partition to execute next at time instant t. Let $\Pi(t)$ denote the partition selected by the global scheduler for t. Then, the tasks of $\Pi(t)$ are scheduled *locally* according to its local scheduling policy.

Each task is associated with a priority, $\texttt{Pri}(\tau_{i,j})$, which can be fixed (e.g., Rate Monotonic (RM) [27] priority assignment) or dynamic (e.g., Earliest Deadline First (EDF) [27]). We do not assume any particular global and local scheduling policies. However, simply for the ease of comprehension of the key

Algorithm 1: `Schedule`($\mathbf{\Pi}, t$)

```
Π(t) ← GlobalScheduler(Π,t) /* ① Selects partition */
if Π(t) ≠ Π(t−1) then
    SuspendPartition(Π(t−1))
end
LocalScheduler(Π(t))            /* ② Selects task */
```

concepts, example schedules in the figures used throughout this paper are based on the fixed-priority global and local schedulings. For most safety-critical systems such as avionics and automotive systems, Rate Monotonic scheduling policy is dominantly employed for local task scheduling due to its stability and conservative predictability [28, 35].

Terminology: Tasks *arrive* to the system on a schedule or in response to external events (e.g., interrupts). For instance, a task can be scheduled to arrive every 100 ms for service of a recurrent workload. The arrival time of task $\tau_{i,j}$ is denoted by $\texttt{t}_\texttt{a}(\tau_{i,j})$. A task is said to be *released* if it becomes visible to the partition to which it belongs, thus becoming available for execution. Each partition Π_i maintains a ready queue Q_i^R of tasks that have been released but not been finished. The Π_i-local scheduler selects a task from $Q_i^R(t)$ for each t.

Each partition $\Pi_i := (T_i, B_i)$ is associated with a non-negative, *maximum budget* B_i; the partition cannot execute for more than B_i (e.g., 20 ms) during a *replenishment period* T_i (e.g., 100 ms). The *remaining* budget for time t is denoted by $B_i(t)$ and $0 \le B_i(t) \le B_i$. No task of Π_i can execute when $B_i(t) = 0$. Partition Π_i is said to be *schedulable* if it is guaranteed to execute for B_i over every replenishment period T_i.

Partition Π_i is said to be *active* at t, denoted by $\texttt{active}(\Pi_i, t)$, if it has a non-zero remaining budget and task(s) to execute, i.e., $B_i(t) > 0$ and $Q_i^R(t) \neq \emptyset$. Only active partitions are subject to the global scheduling.

Definition 1 (Partition preemption)**.** *Partition Π_i is said to be* preempted *if it is not selected by the global scheduler (i.e., $\Pi(t) \neq \Pi_i$) although it has a non-zero remaining budget and has task(s) to run (i.e., it is active). That is,*

$$\texttt{Preempted}(\Pi_i, t) \equiv [\Pi(t) \neq \Pi_i] \wedge \texttt{active}(\Pi_i, t).$$

2.2 Hierarchical Scheduling

Hierarchical scheduling can be categorized into two classes, *static* or *non-static* partitionings, depending on whether partitions are activated at deterministic times or not.

Static Partitioning: Commonly referred to as table-driven scheduling, cyclic-executive, or TDMA (Time Division Multiple Access) scheduling, this approach statically assigns a *fixed* time window of length B_i to each partition Π_i, as shown in Figure 2(a). The fixed schedule repeats every *major cycle* (MC) which is the sum of the partition windows, i.e., $MC = \sum_i B_i$. Hence, each Π_i effectively receives B_i/MC of CPU utilization (e.g., 20 ms/100 ms = 20%). $\Pi(t)$ is deterministic and $\Pi(t) = \Pi(t + MC)$ for any time t. Under the static partitioning scheme, the CPU is left unused if $\Pi(t)$ has no task to

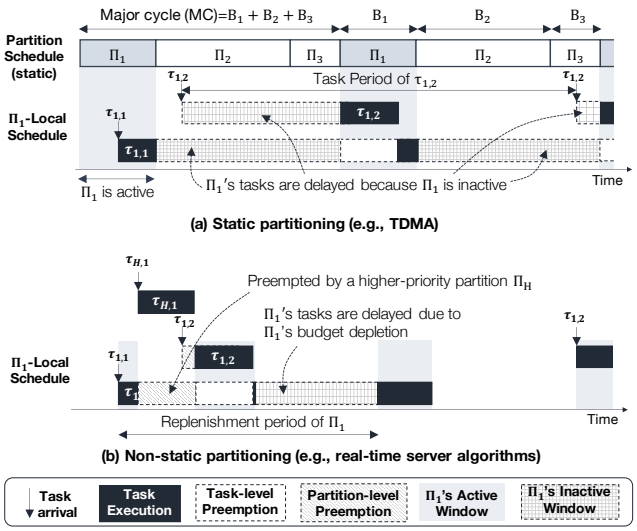


Figure 2: A comparison of static and non-static partitionings.

run even when other partitions have ready tasks. Hence, the temporal behavior of a partition is completely isolated from others. The IMA (Integrated Modular Avionics) architecture of ARINC 653 standard [8] and MILS systems employ this table-driven approach as the partition-level scheduling, and it is implemented in commercial RTOSes such as VxWorks 653 [6] and LynxSecure [1].

The simplicity in the temporal reasoning comes at the cost of inflexibility of resource usage. As shown in Figure 2(a), tasks (e.g., $\tau_{1,2}$) may experience long *initial latency* due to asynchrony between task arrival and the partition's active window. To avoid this, the major cycle could be chosen to be integer multiple of all task periods in the system. However, it is unlikely to find such a major cycle that can remove the initial latencies especially when integrating multiple applications that have different *base rate*, not to mention sporadic (e.g., interrupt-driven) arrivals of tasks. Furthermore, a (system-wide) high-priority task in an inactive partition cannot run until the partition's next window comes, during which lower-priority tasks in other partitions run. These phenomena are inevitable in static partitionings [17, 22].

Non-static Partitioning: This category includes *server*-based approaches such as periodic server [37], sporadic server [39], constant bandwidth server [7], etc. Real-time servers [28, 35] were originally employed to reserve a CPU share for aperiodic tasks while isolating them from critical tasks. In the context of hierarchical scheduling, a server acts as a single partition for a set of tasks that constitutes an application. It is characterized by the budget, B_i, and the replenishment period, T_i. When a task executes, the associated server's budget is depleted for the amount of execution. Each server is assigned a unique priority $\texttt{Pri}(\Pi_i)$, and partitions can be scheduled based on fixed priority (e.g., periodic server, sporadic server) or dynamic priority (e.g., constant bandwidth server).

Different server algorithms differ in the budget consumption and replenishment policies, as shown in Figure 3. For

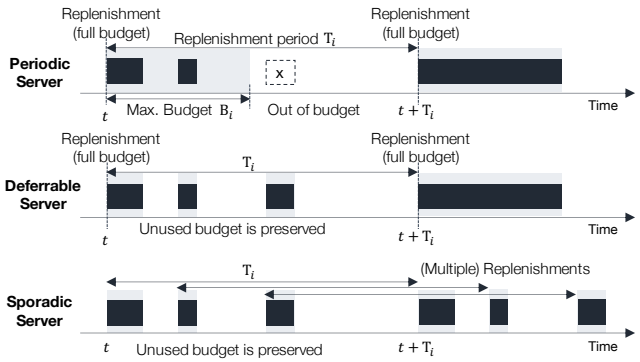


Figure 3: A comparison of real-time server algorithms.

instance, a periodic server [37] is invoked periodically, at which time instant the full budget is replenished. The budget of the current server (i.e., selected by the partition-level scheduler) is consumed even when no task is running. Hence, if a new task arrives after the budget is idled away, the task needs to wait until the next replenishment. In contrast, in deferrable [40] and sporadic server [39] algorithms, budget is consumed only when tasks run. In the former, the full budget is replenished no matter how much budgets are consumed. On the other hand, a sporadic server may have multiple replenishment threads; if a task consumes a budget of b during $[t, t+b)$, the same amount of budget is replenished at $t+T_i$. This effectively limits the server's CPU utilization to B_i/T_i for every T_i time units.

RTOSes implement variants of the server algorithms in consideration of performance and complexity. For instance, the sporadic-polling reservation in LITMUS^{RT} [12] is a variant of the sporadic server – the budget consumption begins once the server takes the CPU and the budget is fully replenished after one period. QNX's adaptive partitioning [2], which enforces CPU utilization of each application partition, also implements a variant of sporadic server in the form of sliding window. Also, Linux (since version 3.14) supports `SCHED_DEADLINE` scheduling policy that can implement constant bandwidth server (CBS) [7] for per-task CPU reservation. Modern real-time hypervisors (e.g., [3, 5]) also support priority-based time-partitioning among virtual machines.

Figure 4 compares task responsiveness under TDMA and sporadic-polling server that are measured from our target system (more detail is provided in Section 6.2 and Appendix B). As the results highlight, and also as discussed above, the

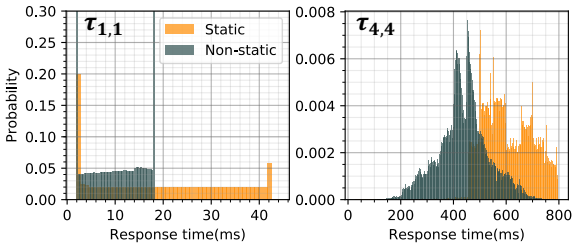
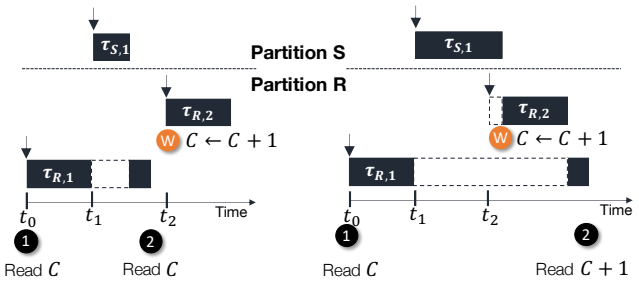


Figure 4: Probability distribution of response times under TDMA and sporadic-polling server schedulers of LITMUS^{RT}.



(a) Π_S sends bit 0 (b) Π_S sends bit 1

Figure 5: $\tau_{R,1}$ and $\tau_{R,2}$ in Π_R can infer Π_S's execution behavior using non-timing information such as a counter C.

non-static partitioning scheme achieves improved average response times compared to the static mechanism (by 108% and 39% for $\tau_{1,1}$ and $\tau_{4,4}$, respectively) mainly because partition executions are not fixed. However, as will be shown in Section 3, such a flexibility is in fact what enables illegitimate information-flow between partitions.

3 Algorithmic Covert Timing-Channel in Hierarchical Scheduling

In this section, we discuss how non-static time partitioning can enable algorithmic covert timing-channels through hierarchical scheduling. Let us first consider Figure 5 with two partitions, Π_S (Sender) and Π_R (Receiver), where $\texttt{Pri}(\Pi_S) > \texttt{Pri}(\Pi_R)$. The receiver partition Π_R has two tasks $\{\tau_{R,1}, \tau_{R,2}\}$ where $\texttt{Pri}(\tau_{R,1}) < \texttt{Pri}(\tau_{R,2})$. A covert communication channel can be formed between the partitions by (i) $\tau_{S,1}$'s varying execution length and (ii) changes in the local schedule of Π_R. In Case (a) of the figure, $\tau_{R,1}$ finishes before $\tau_{R,2}$ starts if $\tau_{S,1}$ executes for a short amount of time, whereas in Case (b) $\tau_{R,1}$ is further delayed by $\tau_{R,2}$ if $\tau_{S,1}$'s execution is long enough. In its simplest form, the sender $\tau_{S,1}$ can have two execution modes (i.e., short or long executions) to encode bits 0 or 1.

Here, Π_R can observe Π_S's varying behavior without using time information. For example, a counter C, shared only between $\tau_{R,1}$ and $\tau_{R,2}$, can be used to infer Π_R's local schedule – $\tau_{R,1}$ checks if the value of C changes from the beginning of its execution (❶) to the end (❷), while $\tau_{R,2}$ increases C by 1 at the beginning of its execution (Ⓦ), as Figure 5 illustrates. The order of ❷ and Ⓦ is influenced by the sender – if Π_S sends 0, $\tau_{R,1}$ will see the counter value remaining same because it finishes before $\tau_{R,2}$ increases C. If $\tau_{R,1}$ sees an increment of C, it indicates that Π_S has signaled bit 1.

To show the vulnerability of existing operating systems to the algorithmic covert timing-channel through hierarchical scheduling, we implemented the scenario described above on LITMUS^{RT} [12], a real-time extension of the Linux kernel. We used its sporadic-polling scheduler, which is a variant of sporadic server. Figure 6 presents the source codes of (i) $\tau_{S,1}$ that varies its execution length to encode bit 0 or 1, (ii) $\tau_{R,1}$ that checks a change in the counter shared with $\tau_{R,2}$, and (iii) $\tau_{R,2}$ that merely increases the shared counter. Tasks run for

```
void sender_job(int bit) {
  if (bit==1)
    n_loops = 10000000;
  else
    n_loops =  2000000;

  for (i=0; i<n_loops; i++)
    asm("nop");
}
```

bit=0: ❶ → ❷ → Ⓦ

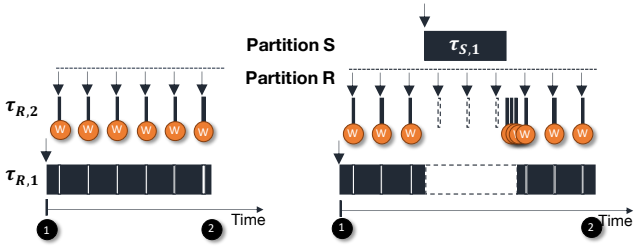
bit=1: ❶ → Ⓦ → ❷

```
int receiver1_job(void) {
  prev_c = shared_c;          ❶
  n_loops = 6000000;

  for (i=0; i<n_loops; i++)
    asm("nop");

  curr_c = shared_c;          ❷
  return curr_c - prev_c;
}
void receiver2_job(void) {
    shared_c++;               Ⓦ
}
```

Figure 6: Implementations of $\tau_{S,1}$, $\tau_{R,1}$, and $\tau_{R,2}$.



(a) curr_c − prev_c = 6 (b) curr_c − prev_c = 9

Figure 7: Extending from Figure 5, $\tau_{R,1}$ can perceive varying amount of preemption.

a pre-defined number of loops (i.e., `n_loops`), thus no time information is needed. The numbers are chosen in such a way that $\tau_{R,1}$ finishes prior to $\tau_{R,2}$'s arrival if $\tau_{S,1}$ sends bit 0. As long as the conditions on the priority relation and the relative phases are met, $\tau_{S,1}$ can send an arbitrary bit-stream to $\tau_{R,1}$. We were also able to reproduce the same scenario on QNX Neutrino as well. We created the partitions using its Adaptive Partitioning Thread Scheduler [2] with the same configuration used in LITMUS^{RT} and C code similar to Figure 6, although there is no priority relation among partitions in this case.

The technique described above can be extended in various ways. Figure 7 shows an extension, in which $\tau_{R,2}$ acts as a regular *tick* counter. While Π_R is preempted by Π_S, $\tau_{R,2}$'s executions are delayed. Because its priority is higher than $\tau_{R,1}$, upon returning from Π_S's preemption, the accumulated jobs of $\tau_{R,2}$ execute, increasing the counter value by the number of times it could not execute during the preemption. Depending on the length of the preemption, the difference `curr_c - prev_c` at ❷ changes, which enables a multi-bit channel.

3.1 Adversary Model

We assume that task execution is time-invariant. That is, if a task executes for a period of time Δt, the progress that it makes from time t_0 to $t_0 + \Delta t$ remains indistinguishably constant even if t_0 changes. This assumption might be violated by, for example, microarchitectural timing-channels [20, 23, 24, 26]. We assume that the microarchitectural timing-channels are properly mitigated [15, 18] to the degree that the time-invariant property holds. We acknowledge that the microarchitectural timing-channels play an important role in interferences. In this paper, we show that an *algorithmic* timing-channel through hierarchical scheduling can exist even if the microarchitec-

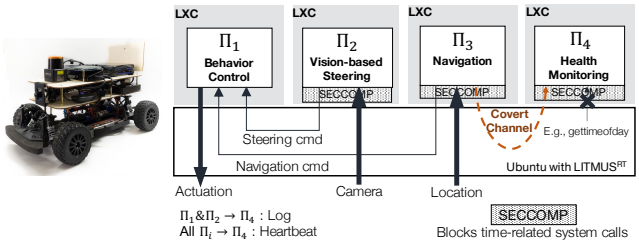


Figure 8: Motivating scenario of illegitimate information flow.

tural timing-channels are completely closed. Hence, we can view the microarchitectural channels as an orthogonal issue.

Partitions can communicate with each other but only through explicit channels to send/receive data such as sensor data, actuation commands, system-monitoring information, and so on. These channels are monitored, and no unknown explicit channels exist. In Linux-based environment, software compartmentalization mechanisms such as Linux containers (LXC) [10] can help reduce hidden channels as well as providing proper resource (e.g., memory, I/O) isolation.

We do not limit an adversary's ability to control its timing; he/she can even request the scheduler to launch tasks at precise times, using facilities that are intended for managing precedence constraints among tasks in both same and different partitions (e.g., data sender and receiver) and also for aligning task arrivals with certain events such as periodic retrieval of sensor data. With such capabilities, the adversary can maximize the chance for successful communication over the covert channel through hierarchical scheduling. Our goal is to prevent the adversary from forming the covert channel even under such optimized conditions.

Motivating Scenario: We implemented the scenario presented above on an 1/10th-scale autonomous driving system that is composed of four partitions as shown in Figure 8. The implementation details are presented in Section 6.1. The partitions are scheduled by the sporadic-polling scheduler, and each partition is isolated inside an LXC. In this system, various run-time information such as driving commands are collected via explicit channels by the health monitoring partition Π_4 for a post-analysis purpose. We can consider an ill-intended system administrator who exploits a covert channel to collect sensitive information such as location data that are supposed not to be collected. In the system, the navigation partition, which computes a navigation path for the behavior controller, may leak the current location data to the health monitoring module in which it is secretly stored along with other run-time information, bypassing communication monitors. Specifically, we took advantage of the watchdog process in Π_4; upon receiving a heartbeat message from Π_3, the receiver task is launched with a delay to time itself to the sender. This approach is advantageous in that the sender and receiver tasks do not need to coordinate their timing in advance – the timing and frequency are controlled by the sender. Of course, if the adversary had a full control of the system, it could be easily configured to launch the tasks in phase.

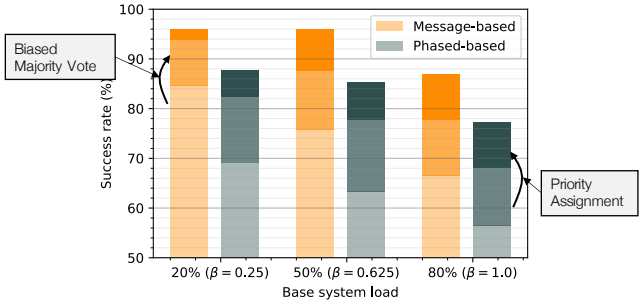


Figure 9: Accuracy of communication over the covert channel through LITMUS^{RT}'s hierarchical scheduling.

3.2 Feasibility Test

Although this paper does not aim to propose an advanced attack technique, we demonstrate the feasibility of the scenarios presented above in a general setting. For this, we used the system configuration shown in Table 1 in Section 6.2. The partitions divide the CPU share equally (with $\alpha = 1.25$). The tasks' inter-arrival times are not fixed, and thus they can arrive at arbitrary times. This creates unpredictable interference with the sender and receiver. We implemented the sender and receiver tasks in Π_3 and Π_4, respectively, as middle-priority tasks and tried three different base system loads ($\beta \in \{0.25, 0.625, 1.0\}$) to vary the amount of noise on the covert channel. The sender and the receiver use a simple repetition code of length 5.

We tried two approaches: (i) phased-based and (ii) message-based launches. In the phase-based approach, the sender and receiver tasks arrive periodically (every 100 ms) from the same initial phase using LITMUS^{RT}'s synchronous-release function. The message-based scheme takes advantage of a legitimate communication channel as explained above. Figure 9 presents the communication accuracy under the two schemes. Each data point is measured for 30 minutes. Although the success rate is considerably high when the system is light-loaded, it drops as the interference by other partitions and tasks increases (i.e., high-loaded). Overall, the message-based coordination achieves higher accuracy than the phase-based scheme because in the former, both the sender and receiver tasks are delayed together until the sender takes the CPU. Whereas in the latter, the receiver's arrival is independent from the sender.

Based on these observations, we tried a simple technique that can significantly improve the accuracies – giving more weight to bit '1' when decoding a repetition code. This is based on the fact that the receiver is more likely to see bit '0' because its execution is likely to be delayed by other tasks (i.e., $\tau_{R,1}$ and $\tau_{R,2}$ in Figure 5(a) are delayed together). As shown in Figure 9, this biased majority voting improves the accuracy by up to 14%. It can be further enhanced if the sender and receiver tasks were allowed to choose their partition-local priorities. This is based on the fact that communication between them is likely to be successful if they execute back-to-back. Hence, we tried giving the Π_3-local lowest-priority to the sender and the Π_4-local highest-priority to the receiver,

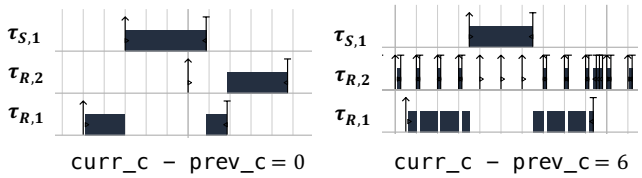


Figure 10: Schedule traces generated by LITMUS^{RT} under our solution that correspond to Figures 5(b) and 7(b).

which resulted in an improvement of up to 9%. Although we did not try, the accuracies can be further improved if, for example, (i) other tasks execute in a strictly-periodic fashion, (ii) the sender and receiver can take into account other tasks' timing properties such as periods, (iii) they can coordinate with their partition-local tasks for a better timing, and so on.

The results above highlight that systems that employ hierarchical scheduling with non-static time partitioning are vulnerable to such covert timing-channel, and an adversary can use various techniques to increase the chance of successful communication. Our solution, which will be presented in later sections, deters such attempts even when the system is configured in favor of the adversary. In fact, the adversary's best chance at distinguishing timing variations from other partitions becomes no better than a random guess because the cases in Figures 5(b) and 7(b) cannot occur under our solution, as shown in Figure 10.

4 Non-interference of Partition-Local Schedule

Following the definition in [32], we define information-flow through hierarchical scheduling as follows:

Definition 2 (Information-flow through hierarchical scheduling)**.** *Information is said to flow from a partition Π_j to another partition Π_i when changes in the temporal behavior of Π_j result in changes in Π_i's observation on its local state.*

Section 3 presented how illegitimate information-flow can be established between non-static partitions even without using time measurements. Specifically, the tasks of partition Π_R were able to perceive Π_S's varying execution behavior from observing changes in their own partition's *local schedule* [29].

Definition 3 (Partition-local schedule)**.** *The Π_i-local schedule is defined as a mapping from the* partition-local time*, $t^{(i)}$, to task set Π_i. The partition-local time advances only when it is selected by the global scheduler to serve its tasks.*

Figure 11 shows how the varying execution of Π_S changes the local schedule of Π_R. For instance, as soon as Π_R returns from the preemption at local time $t_1^{(R)}$, task $\tau_{R,1}$ continues its execution in Case (a). Whereas in Case (b), $\tau_{R,2}$ executes because it is the highest-priority task in the ready queue of Π_R when it resumes. Similarly, at the local time $t_2^{(R)}$, task $\tau_{R,2}$ executes in Case (a) while $\tau_{R,1}$ executes in Case (b).

The fundamental reason why the partition-local schedule changes is because the tasks are released at non-deterministic

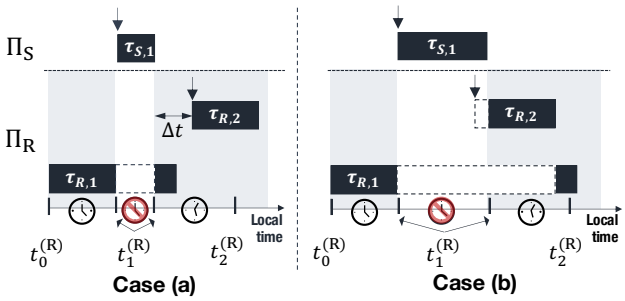


Figure 11: Two different local schedules of Π_R due to the varying execution of Π_S. Notice that the local time of Π_R does not advance while it is not running due to a preemption.

local times even if they arrive at deterministic physical times. For example, $\tau_{R,2}$ arrives at physical time t_2 as shown in Figure 5. However, in the Π_R-local time, it is released at $t_1^{(R)} + \Delta t$ in Case (a) and $t_1^{(R)}$ in (b) as shown in Figure 11. Thus, the amount of progress that $\tau_{R,1}$ makes until $\tau_{R,2}$'s release varies, which enables them to know the order of their executions.

If tasks are released at deterministic local-time points, they always produce the same partition-local schedule because the state of the partition's ready queue changes at the deterministic time points. Following the definition in [29], we define the non-interference property of partition-local schedule as follows:

Definition 4 (Non-interference of partition-local schedule)**.** *Partition Π_i is said to be non-interferent if its local schedule is invariant to how other partitions $\mathbf{\Pi} \setminus \{\Pi_i\}$ execute. Specifically, Π_i's local schedule is deterministic if tasks of Π_i are released at deterministic Π_i-local times.*

5 Partition-Oblivious Hierarchical Scheduling

In this section, we present Blinder, a run-time schedule transformation algorithm that makes non-static time partitions oblivious to others' varying behavior. Our design goals for Blinder are to (i) make it agnostic to global and local scheduling mechanisms, (ii) incur low scheduling overhead, and (iii) make its worst-case impact on the responsiveness deterministic, which is important for real-time applications.

5.1 High-level Idea

Partition-local schedules can be made deterministic simply by a static partitioning; that is, partitions are suspended at deterministic time points for fixed amount of time. This, however, may lead to low CPU utilization as briefly discussed in Section 2.2. Hence, instead of controlling when and how long partitions should execute, we aim to allow partitions to be scheduled still in non-static ways (thus taking advantage of the improved responsiveness of non-static partitioning schemes) while preventing the non-deterministic behaviors from being distinguishable by local tasks. Blinder achieves this by controlling when to introduce task to partition, i.e., *task release time*.

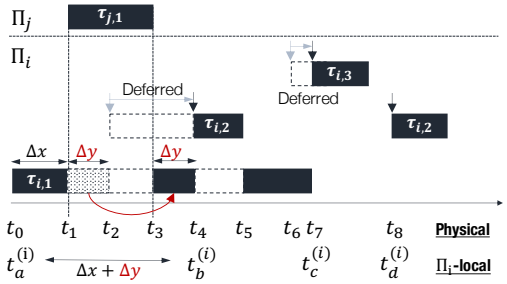


Figure 12: Blinder controls the release times of newly-arrived tasks to make the partition-local schedule deterministic.

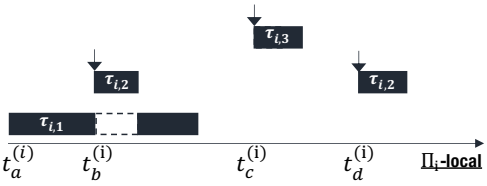


Figure 13: Π_i's canonical local schedule.

Figure 12 illustrates the high-level idea using a two-partition example that is similar to the case considered in Figure 5. Here, the release of $\tau_{i,2}$ that arrives at physical time t_2 is deferred *as if* Π_j*'s preemption did not occur*. Specifically, $\tau_{i,2}$'s release is deferred until t_4 to ensure that $\tau_{i,1}$ would execute for the amount of time that it would have done (i.e., $\Delta y = t_2 - t_1$) if the preemption by Π_j did not happen. Hence, $\tau_{i,2}$ is released at $t_4 = t_3 + \Delta y$ where t_3 is when Π_i returns from the preemption. Thus, $\tau_{i,1}$ is guaranteed to execute for

$$(t_1 - t_0) + (t_4 - t_3) = (t_1 - t_0) + (t_2 - t_1) = t_2 - t_0 = \Delta x + \Delta y.$$

Note that it is independent from the duration of the preemption by Π_j, i.e., $t_3 - t_1$. Thus, $\tau_{i,1}$ makes the same amount of progress, i.e., $\Delta x + \Delta y$, until $\tau_{i,2}$ is released, regardless of how long the preemption is. Such a deferment is applied also to certain tasks that arrive while the partition is active. $\tau_{i,3}$ is an example. If it was released immediately upon the arrival at time t_6, $\tau_{i,1}$ would be preempted by $\tau_{i,3}$, which would not occur if Π_j's preemption was shorter or did not happen. On the other hand, $\tau_{i,2}$'s second arrival at time t_8 does not need to be deferred because all the prior executions that are *affected* by the partition-level preemption complete before t_8, and thus $\tau_{i,2}$'s execution does not change the local schedule of Π_i.

Blinder guarantees that tasks are always released at deterministic partition-local times by enforcing that the partition-local schedules to follow the partition's *canonical local schedule* – the local schedule when no other partitions run [29]. Figure 13 shows an example canonical local schedule of Π_i that resulted in the schedule in Figure 12. For instance, the first arrival of $\tau_{i,2}$ is released at Π_i-local time $t_b^{(i)} = t_a^{(i)} + (\Delta x + \Delta y)$. Only the corresponding physical time varies in the presence of other partitions. An actual schedule observed in the physical-time domain under Blinder can be viewed as the canonical schedule being stretched out over time by higher-priority partitions. Hence, actual schedules vary with the partition schedule while the partition-local schedule remains same.

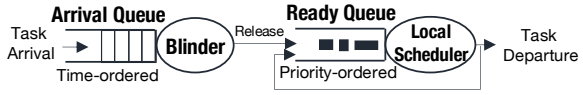


Figure 14: Blinder uses arrival queue to control task release.

The canonical local schedule, however, cannot be statically constructed because tasks may arrive at arbitrary times and have variable execution times. Most importantly, these in turn affect when partition budget is depleted and replenished. Hence, the challenge lies in determining for how long $\tau_{i,2}$'s release must be deferred (i.e., Δy in Figure 12), which depends on the amount of CPU time that $\tau_{i,1}$ would have used if not preempted by Π_j. The crux of Blinder algorithm is the *on-line* construction of canonical local schedule.

5.2 Blinder Algorithm

Figure 14 illustrates a high-level overview of Blinder algorithm. It constructs a canonical local schedule of partition in the run-time by having an *arrival queue* hold newly-arrived tasks until releasing them to the ready queue at the right timing: it depends on how the partition is scheduled. Without Blinder, every newly-arriving task is immediately inserted to the ready queue. As discussed earlier, this is the very source of information-flow between partitions.

5.2.1 Scheduling Primitives

Blinder algorithm does not require a change in the generic hierarchical scheduler presented in Algorithm 1 (Section 2.1). That is, at each scheduling decision, a partition is selected according to the global scheduling policy. It is important to notice that Blinder only controls the task release times. Once tasks are released, they are scheduled according to the partition-specific local scheduling policy that is independent from Blinder algorithm, as shown in Algorithm 2 (last line).

Task arrival, release, and departure: `TaskArrive` is invoked at any time when a new task $\tau_{i,j}$ arrives to partition Π_i. The task is inserted into the arrival queue $Q_i^A(t)$, annotated with the arrival time $\mathtt{t_a}(\tau_{i,j}) = t$, as shown in Algorithm 3. As long as the partition Π_i is selected by the global scheduler to run, i.e., $\Pi(t) = \Pi_i$, it checks for task release. As shown in Algorithm 2, Blinder releases tasks that meet a certain condition (which will be introduced shortly) by moving them from the arrival queue to the ready queue. `TaskDepart`, shown in

Algorithm 2: `LocalScheduler`(Π_i)

Δt_i: time used by Π_i since the last check
$\mathtt{Used}_i \leftarrow \mathtt{Used}_i + \Delta t_i$
for $\tau_{i,j} \in Q_i^A(t)$ **do**
  $\mathtt{lag}_{i,j} \leftarrow \mathtt{lag}_{i,j} - \Delta t_i$ `/* Reduce lag */`
  if $\mathtt{lag}_{i,j} == 0$ **then**
    $Q_i^A(t) \leftarrow Q_i^A(t) - \{\tau_{i,j}\}$
    $Q_i^R(t) \leftarrow Q_i^R(t) \cup \{\tau_{i,j}\}$ `/* Release task */`
  end
end
$\tau_i(t) \leftarrow$ select a task from $Q_i^R(t)$ according to local scheduling policy

Algorithm 3: TaskArrive($\tau_{i,j}$)

$\mathtt{t_a}(\tau_{i,j}) \leftarrow t$ /* Arrival time of $\tau_{i,j}$ */
$Q_i^A(t) \leftarrow Q_i^A(t) \cup \{\tau_{i,j}\}$
$\mathtt{lag}_{i,j} \leftarrow \mathtt{lag}_i(t)$ /* Initialize $\mathtt{lag}_{i,j}$ */
if $\mathcal{M}_i(t) == \mathtt{M_{normal}}$ *and* $\mathtt{Preempted}(\Pi_i, t) == \mathtt{True}$ **then**
  $\mathtt{EnterDeferredMode}(\Pi_i)$
end

Algorithm 4: TaskDepart($\tau_{i,j}$)

$Q_i^R(t) \leftarrow Q_i^R(t) - \{\tau_{i,j}\}$
if $Q_i^R(t) == \emptyset$ *and* $\mathcal{M}_i(t) == \mathtt{M_{deffered}}$ **then**
  if $Q_i^A(t) == \emptyset$ **then**
    $\mathcal{M}_i(t) \leftarrow \mathtt{M_{normal}}$ /* Switch to *normal* mode */
  else
    $\mathtt{ShiftRelease}(\Pi_i)$ /* Update lag in $Q_i^A(t)$ */
  end
end

Algorithm 4, is called when a task execution completes and removes the task from the ready queue.

Task release mode: Blinder defines the *task release mode* of partition Π_i at time t, $\mathcal{M}_i(t)$: *normal* ($\mathtt{M_{normal}}$) or *deferred* ($\mathtt{M_{deffered}}$) release modes. While a partition is in the normal release mode, any newly-arriving task bypasses the arrival queue and thus immediately becomes ready to execute. If the partition is in the deferred release mode, the task is held in the arrival queue until it is released by Blinder. Each partition is initialized to the normal release mode. If a partition remains in the normal release mode, its behavior is identical to what it would be in a system without Blinder.

Entering into deferred release mode: A partition Π_i enters into the deferred release mode when a preemption on Π_i by another partition explicitly begins, as shown in Algorithm 5. Π_i is said to be *implicitly* preempted if it becomes active (i.e., it has a non-zero remaining budget and tasks to run, as defined in Section 2.1) due to a new task arrival or a budget replenishment, but it is not selected by the global scheduler. Hence, the partition can also enter into the deferred release mode by a task arrival (as shown in Algorithm 3) or a budget replenishment. Note that a suspension due to budget depletion does not change the task release mode.

5.2.2 Lag-based Task Release

As discussed in Section 5.1, task releases are deferred in order to hide preemptions by other partitions from local tasks. Hence, release times are affected by when and how long preemptions occur. Let us consider Figure 12 again. Suppose that Π_i has not been preempted by any partition prior to t_1. That is, t_1 is the time instant at which the local schedule of Π_i *starts deviating from* its canonical schedule shown in Figure 13. From this moment, the progresses of the tasks in the actual schedule (i.e., Figure 12) *lag behind* those in the canonical schedule. Hence, when a new task arrives after t_1, its release should be deferred until the actual schedule has caught up to

Algorithm 5: SuspendPartition(Π_i)

if $\mathcal{M}_i(t) == \mathtt{M_{normal}}$ *and* $\mathtt{Preempted}(\Pi_i, t) == \mathtt{True}$ **then**
  $\mathtt{EnterDeferredMode}(\Pi_i)$
end

Algorithm 6: EnterDeferredMode(Π_i)

$\mathcal{M}_i(t) \leftarrow \mathtt{M_{deffered}}$
$\mathtt{Used}_i \leftarrow 0$
$\mathtt{t_{def}}(\Pi_i) \leftarrow t$ /* Current time */
$\mathtt{B_{def}}(\Pi_i) \leftarrow B_i(t)$ /* Remaining budget */

the progress that the partition would have made until the task is released in the canonical counterpart.

Since the canonical schedule cannot be statically determined in advance, Blinder constructs an *imaginary* canonical schedule on the fly and updates it upon certain scheduling events. For this, Blinder keeps track of the following per-partition information to determine task release times:

Definition 5 (Available time)**.** $\mathtt{available}_i(t)$ *is the maximum amount of time that would have been available to the tasks of* Π_i *until time* t *since the latest time instant at which* Π_i *entered into the deferred release mode.*

Definition 6 (Used time)**.** $\mathtt{used}_i$ *is the amount of time that has actually been used by* Π_i *since the latest time instant at which* Π_i *entered into the deferred release mode.*

Note that $\mathtt{used}_i \leq \mathtt{available}_i(t)$ always hold as $\mathtt{used}_i$ remains same as long as Π_i is suspended due to a preemption. $\mathtt{used}_i$ depends on the partition-level schedule and it can be easily kept track of by counting the partition active times, as done in Algorithm 2. For instance, in Figure 12, at time t_6, $\mathtt{used}_i = t_6 - t_3$. On the other hand, the computation of $\mathtt{available}_i(t)$ is not straightforward because it depends on the budget constraints, as we will detail shortly. In the example above, we simply assumed no limitation on partition budgets. Hence, $\mathtt{available}_i(t)$ was always the relative offset of t from the beginning of the current deferred release mode, e.g., $\mathtt{available}_i(t_6) = t_6 - t_1$.

Now, we define the *lag* as the difference between $\mathtt{available}_i(t)$ and $\mathtt{used}_i$:

Definition 7 (Lag)**.** $\mathtt{lag}_i(t)$ *is the maximum amount of time by which the current local schedule of* Π_i *at time* t *lags behind the canonical schedule. It is computed by*

$$\mathtt{lag}_i(t) = \mathtt{available}_i(t) - \mathtt{used}_i.$$

A positive $\mathtt{lag}_i(t)$, say l, when task $\tau_{i,j}$ arrives at time t means that Π_i would have executed up to the amount of l until $\tau_{i,j}$ arrives if no preemptions on Π_i have occurred. Hence, the release of $\tau_{i,j}$ should be deferred until Π_i will have executed for l. This guarantees the tasks of Π_i to make the same amount of progresses that they would have made in the canonical schedule until $\tau_{i,j}$'s release. Therefore, when a new task $\tau_{i,j}$ arrives (Algorithm 3), Blinder initializes the per-task

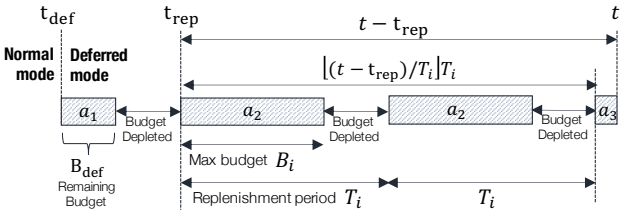


Figure 15: Maximum available time for Π_i from the beginning of the deferred release mode until an arbitrary time t.

lag value, $\texttt{lag}_{i,j}$. Then, it reduces the `lag` values of the tasks in the arrival queue as long as the partition runs and releases those tasks whose `lag` value become zero (Algorithm 2). Note that `lag` is always zero in the normal release mode.

Budget constraint and available time: So far, it has been assumed that partitions have unlimited budgets, and thus the whole period of preemption on a partition was assumed to be fully available to the partition if preemptions did not happen. However, the budget constraint could have limited the available time to the partition. Suppose Π_i enters into the deferred release mode at time $\texttt{t}_{\texttt{def}}$. Then, the maximum amount of time available to the partition from $\texttt{t}_{\texttt{def}}$ to an arbitrary time instant t in the canonical local schedule (i.e., when no preemptions occur) is composed of the following terms (shown in Figure 15):

- Remaining budget until the next replenishment or t:

$$\texttt{a}_1 = \min\left[\texttt{B}_{\texttt{def}}, (t - \texttt{t}_{\texttt{def}}), (\texttt{t}_{\texttt{rep}} - \texttt{t}_{\texttt{def}})\right]$$

- Full-budget replenishments:

$$\texttt{a}_2 = \lfloor (t - \texttt{t}_{\texttt{rep}})/T_i \rfloor B_i$$

- Last replenishment:

$$\texttt{a}_3 = \min\left[B_i, (t - \texttt{t}_{\texttt{rep}}) - \lfloor (t - \texttt{t}_{\texttt{rep}})/T_i \rfloor T_i\right]$$

Here, $\texttt{B}_{\texttt{def}}$ is the remaining budget at the time of entering into the deferred release mode, and $\texttt{t}_{\texttt{rep}}$ is when the next replenishment is scheduled at, both of which are known at $\texttt{t}_{\texttt{def}}$. Then, $\texttt{available}_i(t)$ is computed as follows:

$$\texttt{available}_i(t) = \texttt{a}_1 + \texttt{a}_2 + \texttt{a}_3,$$

where $\texttt{a}_2 = \texttt{a}_3 = 0$ if $t < \texttt{t}_{\texttt{rep}}$. That is, the maximum available time is obtained by assuming that all budgets are *used up as soon as* they become available.

The maximum available time depends also on the budget replenishment policy. In the formulation above, we assumed a fixed-replenishment policy that can be found from periodic and deferrable servers; the budget is replenished to the maximum budget at a fixed-interval no matter how budgets are used. Hence, the future replenishment times are always known. In the other category, such as sporadic server, there could exist multiple replenishment points determined by when and how much budgets have been used. Nevertheless, the maximum available time can still be obtained, even in such a case, because the information needed is available at the time of entering into the deferred release mode; the `available` time is decomposed into smaller `available` times, each of which can be computed by the formula above. Blinder algorithm can be applied to other hierarchical scheduling schemes as long as the `available` times can be deterministically computed.

Algorithm 7: `ShiftRelease`(Π_i)

```
τ_{i,x} ← earliest arrival in Q_i^A(t)
B_def(Π_i) ← B_i(t_a(τ_{i,x})) (See Appendix A)
t_def(Π_i) ← t_a(τ_{i,x})
Used_i ← 0
for τ_{i,j} ∈ Q_i^A(t) do
    lag_{i,j} ← lag_i(t_a(τ_{i,j}))          /* Update lag */
end
```

Lag-update: The `available` function assumes the *maximal* use of budget, subject to the budget constraint, over a time interval. However, the actual execution may be smaller than what is assumed by `available` function, which leads to a situation in which the partition becomes idle while some tasks are held in the arrival queue. One can improve their responsiveness by releasing them *earlier* than originally projected. This can be done by reducing their `lag` values, as if time is *fast-forwarded*. Suppose the ready queue becomes empty at time t. The `lag` value for each $\tau_{i,j} \in Q_i^A(t)$ is adjusted by `ShiftRelease` in Algorithm 7 in such a way that the earliest arrival among the tasks in $Q_i^A(t)$, denoted by $\tau_{i,x}$, is released immediately at t. That is, the beginning of the deferred release mode is reset to $\tau_{i,x}$'s arrival, i.e., $\texttt{t}_\texttt{a}(\tau_{i,x})$. To update the `lag` values of the tasks in the arrival queue, the remaining budget of Π_i at $\tau_{i,x}$'s arrival (i.e., the new beginning of the deferred release mode) should be computed. Appendix A explains how to compute $B_i(\texttt{t}_\texttt{a}(\tau_{i,x}))$ from $\texttt{lag}_{i,x}$ at time t.

Switching to normal release mode: If (i) the ready queue becomes empty and (ii) there is no pending task-release (i.e., the arrival queue is empty), the partition switches to the normal release mode, as shown in Algorithm 4. It is when all the tasks that have arrived before or during the deferred release mode complete their executions, resulting in the local schedule being synchronized with the canonical one.

Example: Let us consider two partitions Π_H and Π_L that consist of 1 and 3 tasks, respectively, with the following configuration: Π_L =(10 ms, 7 ms), $\texttt{Pri}(\Pi_H) > \texttt{Pri}(\Pi_L)$, and $\texttt{Pri}(\tau_{L,1}) < \texttt{Pri}(\tau_{L,2}) < \texttt{Pri}(\tau_{L,3})$. Figure 16 shows the schedule trace generated by LITMUS$^{\texttt{RT}}$. Suppose we are interested in determining the release time of $\tau_{L,2}$ that arrives at $t = 21$ ms.

1. Π_L enters into the deferred release mode at $\texttt{t}_{\texttt{def}} = 15$ ms due to the partition-level preemption by Π_H.

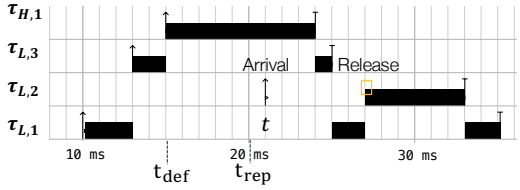


Figure 16: Example schedule trace generated by LITMUS$^{\texttt{RT}}$.

2. The remaining budget of Π_L, $\mathtt{B_{def}}$, is 2 ms because 5 ms has been used by $\tau_{L,1}$ and $\tau_{L,3}$ since time 10 ms.
3. Because Π_L started consuming the budget at time 10 ms, the next replenishment time is $\mathtt{t_{rep}} = 20$ ms. From then to $t = 21$ ms, a budget of 1 ms is also available to Π_L. Hence, $\mathtt{available}_L(21) = 2+1 = 3$ ms.
4. From $\mathtt{t_{def}}$, Π_L has never executed, thus $\mathtt{used}_L = 0$ at $t = 21$. This results in $\mathtt{lag}_{L,2} = \mathtt{lag}_L(21) = 3$ ms.
5. Π_L returns from the preemption by Π_H at time 24 ms. $\tau_{L,3}$ resumes its execution because it is the highest-priority task in the ready queue of Π_L. After $\tau_{L,3}$ finishes, $\tau_{L,1}$ resumes its execution.
6. After 3 ms from the Π_L's resumption, $\mathtt{lag}_{L,2}$ becomes 0. Hence, $\tau_{L,2}$ is released at time 27 ms.

Proof of non-interference: The following theorem proves the non-interference property (Definition 4) of schedules transformed by Blinder algorithm.

Theorem 1. *With Blinder algorithm, tasks are released at the deterministic partition-local times.*

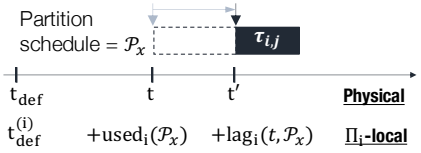


Figure 17: Irrespective of the partition-level schedule $\mathcal{P}_x$, task $\tau_{i,j}$ is released at the deterministic Π_i-local time.

Proof. Suppose at time $\mathtt{t_{def}}$, partition Π_i enters into the deferred release mode. Let us consider two arbitrary partition-level schedules $\mathcal{P}_1$ and $\mathcal{P}_2$ that are equivalent until time $\mathtt{t_{def}}$. Let $\mathtt{t}^{(i)}_{\mathtt{def}}$ be the corresponding Π_i-local time, as depicted in Figure 17. Now, consider a task $\tau_{i,j}$ that arrives at an arbitrary (physical) time $t \geq \mathtt{t_{def}}$. The two different partition schedules during $[\mathtt{t_{def}}, t)$ result in different $\mathtt{lag}$ values, $\mathtt{lag}_i(t,\mathcal{P}_1)$ and $\mathtt{lag}_i(t,\mathcal{P}_2)$, for $\tau_{i,j}$. By the definition, $\mathtt{lag}_i(t,\mathcal{P}_x) = \mathtt{available}_i(t,\mathcal{P}_x) - \mathtt{used}_i(\mathcal{P}_x)$ where both $\mathtt{available}$ and $\mathtt{used}$ times are parameterized by partition schedule $\mathcal{P}_x$. Note that $\mathtt{available}_i(t,\mathcal{P}_x)$ is invariant to $\mathcal{P}_x$ because it computes the maximum amount of time available to Π_i when no other partitions run. Hence, $\mathtt{available}_i(t,\mathcal{P}_1) = \mathtt{available}_i(t,\mathcal{P}_2)$, which leads to

$$\mathtt{used}_i(\mathcal{P}_1) + \mathtt{lag}_i(t,\mathcal{P}_1) = \mathtt{used}_i(\mathcal{P}_2) + \mathtt{lag}_i(t,\mathcal{P}_2). \quad (1)$$

By the definition of the $\mathtt{used}$ time, the Π_i-local time at which $\tau_{i,j}$ arrives is $\mathtt{t}^{(i)}_{\mathtt{def}} + \mathtt{used}_i(\mathcal{P}_1)$ for $\mathcal{P}_1$. Then, $\tau_{i,j}$ is released after the partition executes for $\mathtt{lag}_i(t,\mathcal{P}_1)$, which results in the release time of $\mathtt{t}^{(i)}_{\mathtt{def}} + \mathtt{used}_i(\mathcal{P}_1) + \mathtt{lag}_i(t,\mathcal{P}_1)$. Similarly, $\tau_{i,j}$ is released at $\mathtt{t}^{(i)}_{\mathtt{def}} + \mathtt{used}_i(\mathcal{P}_2) + \mathtt{lag}_i(t,\mathcal{P}_2)$ for $\mathcal{P}_2$. Then,

$$\mathtt{t}^{(i)}_{\mathtt{def}} + \mathtt{used}_i(\mathcal{P}_1) + \mathtt{lag}_i(t,\mathcal{P}_1) = \mathtt{t}^{(i)}_{\mathtt{def}} + \mathtt{used}_i(\mathcal{P}_2) + \mathtt{lag}_i(t,\mathcal{P}_2),$$

due to Eq. (1). That is, the release time of $\tau_{i,j}$ in Π_i-local time is same no matter what the partition-level schedule $\mathcal{P}_x$ is. □

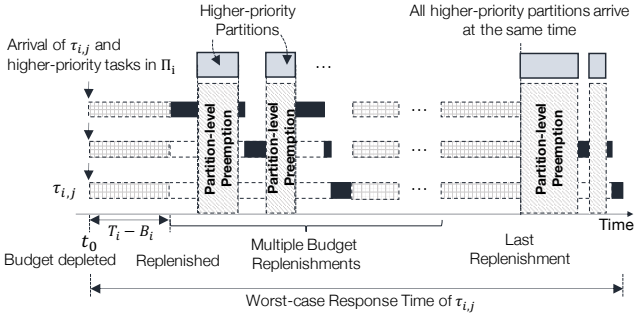


Figure 18: Worst-case response time of $\tau_{i,j}$.

5.3 Schedulability Analysis

In this section, we analyze the schedulability of real-time tasks under Blinder. The schedulability is tightly dependent on particular choices of global and local scheduling policies as well as budget replenishment policy. Hence, we base our analysis on the fixed-priority server algorithm of LITMUS$^{\mathrm{RT}}$ on which our implementation is based.

Let us first consider the case without Blinder. The worst-case situation for $\tau_{i,j}$ (illustrated in Figure 18) is when (a) it arrives at time t_0, at which the partition Π_i's budget is depleted by lower-priority tasks as soon as possible, all the higher-priority tasks in the same partition arrive together at t_0, and their subsequent invocations arrive as frequently as possible; (b) the local task executions are served by Π_i over one or multiple replenishment periods (at most B_i over T_i); (c) all partitions that can preempt Π_i are replenished to their full budgets together when Π_i's last replenishment happens, and they execute as maximally and frequently as possible, thus maximally delaying $\tau_{i,j}$'s remaining executions [14].

In fact, the above worst-case situation also holds for Blinder. In other words, Blinder does not increase the worst-case response time (WCRT) of tasks as long as partitions are *schedulable*; a partition Π_i is said to be schedulable if it is guaranteed to execute for B_i over every replenishment period T_i. By viewing partitions as periodic tasks, the following iterative equation [21] can be used to check if Π_i is schedulable:

$$w_i^{n+1} = B_i + \sum_{\mathtt{Pri}(\Pi_k) > \mathtt{Pri}(\Pi_i)} \lceil w_i^n / T_k \rceil B_k, \quad (2)$$

where $w_i^0 = B_i$. If w_i^n converges and is upper-bounded by the replenishment period T_i, Π_i is guaranteed to serve B_i to its tasks over every period T_i, due to the critical instant theorem [27]. As an example, Π_4 in Table 1 in Section 6.2 takes up to 38 ms to execute for $B_4 = 10$ ms. If we increase the budgets for all partitions by 25%, Π_4 becomes unschedulable.

Lemma 1. $\mathtt{lag}_i \leq B_i$ *always holds if* Π_i *is schedulable.*

Proof. Suppose Π_i is being replenished to its full budget B_i at time t and all partitions that can preempt Π_i are replenished together and use up their budgets as frequently and maximally as possible. If there was no such preemption on Π_i, it could have served its tasks for up to B_i during $[t, t+T_i)$. Hence, $\mathtt{available}_i$ can reach up to B_i. On the other hand,

$\texttt{used}_i$ remains 0 until Π_i takes the CPU. Therefore, $\texttt{lag}_i$ can reach up to B_i in the first period. From then on, over each replenishment period the partition is provided B_i of available time that it can fully use during the period as it is schedulable. Therefore, due to $\Delta\texttt{lag}_i = \Delta\texttt{available}_i - \Delta\texttt{used}_i = 0$, $\texttt{lag}_i$ cannot grow over B_i. In fact, $\texttt{lag}_i$ is always reduced to 0 by the next-replenishment time. □

Theorem 2. *If partition Π_i is schedulable, Blinder does not increase the worst-case response times of its tasks.*

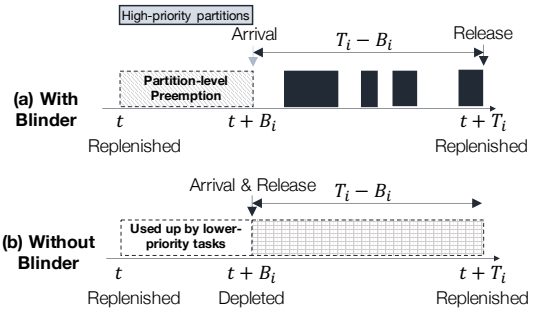


Figure 19: Worst-case initial latency remains same.

Proof. This can be proved by showing that the worst-case initial latency is same whether or not Blinder is used. Suppose Π_i is replenished to its full budget at time t, as shown in Figure 19. The maximum $\texttt{lag}_i$ is achieved when Π_i's execution is delayed by higher-priority partitions at least until time $t + B_i$, as shown in Case (a), at which $\texttt{lag}_i$ becomes B_i. Now, due to Lemma 1, $\texttt{lag}_i$ returns to 0 by or earlier than the end of the current period $t + T_i$. Hence, in the worst-case, a task $\tau_{i,j}$ that arrives at time $t + B_i$ can have the initial latency of $T_i - B_i$. Without Blinder, the worst-case situation for $\tau_{i,j}$ occurs when it arrives as soon as Π_i's budget is depleted, which can happen as early as at time $t + B_i$. As shown in Case (b), the worst-case initial latency is $T_i - B_i$. Therefore, $\tau_{i,j}$ can start its execution at earliest at $t + T_i$. From then on, its execution is independent from whether Blinder is used or not. □

If Π_i is not schedulable due to an ill-configuration of partition parameters, the $\texttt{lag}_i$ might not be bounded if, albeit unlikely, Π_i's workload is infinite, because $\Delta\texttt{lag}_i$ over certain replenishment periods could be non-zero. In practice, it is uncommon to assign parameters that make partitions unschedulable. System integrator can use the exact analysis presented in Eq. (2) for the test of partition schedulability.

Average-case response times: It is worth noting that, as will be experimentally shown in Section 6.2, high-priority tasks tend to experience longer *average-case* response times due to the lag-based task release. However, because the use of lag does not change both the demand from tasks and the supply from partition, increases in the average-case response times of high-priority tasks is compensated by decreases in low-priority tasks' response times. Also, the impact on response times tends to be higher for low-priority partitions as they are more likely to operate in the deferred release mode than high-priority partitions.

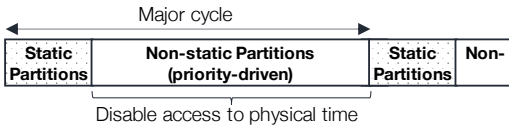


Figure 20: Mix of static and non-static time partitioning.

5.4 Discussion

Physical time: If the precise physical time is available to tasks in the receiver partition (i.e., Π_R in Figure 5), they can directly perceive changes in their timings. Certain countermeasures such as fuzzy time [19, 42] and virtual time [25, 45] techniques can mitigate the threat. However, partitions under a hierarchical scheduling can still form an algorithmic covert-channel without time information as presented in Section 3.

Certain applications still require the physical time information, making the above mitigations inapplicable. One possible way to prevent such applications from exploiting the hierarchical scheduling is to serve them in static partitions. As shown in Figure 20, a system can host a mix of static and non-static partitions by (i) allocating fixed time intervals for the static partitions and (ii) letting the others (i.e., non-static partitions) compete for the remaining times based on their priorities. Note that there can be multiple static partitions placed anywhere in the major cycle. From the perspective of the non-static partitions, the static partitions can be viewed as another non-static partition with the highest-priority and the replenishment period being equal to the major cycle. The non-static partitions are subject to Blinder, and access to precise physical time is disabled for their user processes. In our prototype, we use a sandbox-based method to block physical-time access for a demonstration purpose.

For most real-time applications, their correct functioning highly depends on the execution regularity and schedulability. Physical control processes use time elapsed between successive invocations to estimate a change in the process state over time. For low-dynamics systems, the interval can be approximated by a constant time resolution. In such cases, applications do not need precise physical times. In Section 6.1, we discuss the practicality of such an approximation based on our prototype implementation. On the other hand, high-dynamics processes require precise time information and such tasks can be served in static partitions as explained above. In fact, such highly critical applications are less likely to be malicious as they tend to have low complexity and go through a stringent verification process due to safety requirements.

Modularity: Blinder is modular in that it can be enabled/disabled independently for each partition because it does not change a partition's behavior from others' points of view. Accordingly, enabling/disabling Blinder for individual partition does not affect others' schedulability. This modularity is especially useful when certain partitions are verified to be trustworthy, and such applications are free of accessing precise physical time.

Algorithm complexity: In the normal release mode `lag` is not computed, and the arrival queue is always empty. Hence, Blinder does not enter the loop in `LocalScheduler` (Algorithm 2). Therefore, the algorithm complexity in this mode is $O(1)$. In the deferred release mode, both `LocalScheduler` and `ShiftRelease` update the `lag` values of the tasks in the arrival queue. Because a `lag` calculation is a constant-time operation, the worst-case complexity for each partition is linear to the size of its task set. Note that the operations are independent from the number of partitions in the system. Hence, letting M be the total number of tasks in the system, the asymptotic complexity is $O(M)$.

Multicore: Blinder can be applied to a multicore processor without any modification. This feature is only disadvantageous to adversaries–especially if partitions can migrate between cores. This is because the sender partition may not be able to preempt the receiver partition. If migration is not allowed, Blinder can be independently applied to each of the application sets that share a CPU core. However, if a partition can run multiple tasks simultaneously across multiple cores, one of them can serve as an implicit clock (e.g., using a shared counter in a tight loop). Hence, partitions should not be allowed to occupy multiple cores at the same time, as well as disallowing shared memory and messaging across cores. In fact, in high-assurance systems, it is a common practice to fix a partition to a core to minimize the unpredictability caused by concurrency [36].

6 Evaluation

6.1 Use Case

Platform: We applied Blinder to an 1/10th-scale self-driving car platform. Figure 8 in Section 3.1 presented the overall system architecture. It autonomously navigates an indoor track using a vision-based steering control and an indoor positioning system. The health monitoring partition, Π_4, collects runtime log data and also runs watchdog process that monitors the application partitions' heartbeat messages.

Blinder Implementation: We implemented Blinder in the latest version of LITMUS^{RT} (based on Ubuntu 16.04 with kernel version 4.9.30) which we run on Intel NUC mini PC that has Intel Core i5-7260U processor operating at 2.20 GHz and a main memory of 8 GB. Our implementation is based on LITMUS^{RT}'s partitioned reservation (`P-RES`) framework. For our evaluation, we applied Blinder to the sporadic-polling server of `P-RES` which is a variant of the sporadic-server algorithm; the full budget is replenished after one period from the beginning of first consumption, instead of maintaining multiple replenishment threads. Hence the same `available` function presented in Section 5.2.2 is used because only one $\mathtt{t}_{\mathtt{rep}}$ (i.e., the next replenishment time) exists and is known at any time instant. We used the fixed-priority preemptive scheduling for the partition-local scheduling. Our implementation is denoted by `P-RES-NI` (`P-RES` with *non-interference*).

One implementation challenge that needs to be highlighted is that LITMUS^{RT} reservation does not have a clock to generate regular tick-based signals. It rather uses a Linux high-resolution timer (hrtimer) [16] to set an absolute expiration time instant for the next schedule. Every time when the scheduler returns to the user thread, it resets this timer to the closest instant that needs to be responded by the scheduler. In `P-RES`, the next reschedule point is determined by the minimal value of the time slices of the local scheduler, the remaining budget, and the next replenishment time. For `P-RES-NI`, we added one more condition, that is the remaining `lag` for the head of the arrival queue, to accurately schedule task release points.

Blocking access to physical time: The behavior controller partition, Π_1, is allowed to access the physical time because it is given the highest priority. The other partitions do not need the precise physical time information. Thus, we blocked them from accessing the physical time. Specifically, we implemented a secure launcher that creates a restricted execution environment for user tasks based on seccomp (secure computing mode) [11] that can filter any system calls. We blacklisted time-related system calls such as `time`, `clock_gettime`, `timer_gettime`, etc. In addition, we dropped the permissions that could leak physical time information, including time-relevant devices (e.g., `/dev/hpet`), time-stamp counter (`TSC`), model-specific registers, and virtual ELF dynamic shared object (`vDSO`) [13]. The tasks of Π_2, Π_3, and Π_4 are launched by this secure launcher. Other approaches such as fuzzy time [19, 42] and virtual time [25, 45] could also be used for them.

The navigation partition also runs a Kalman filter-based localization task that requires a time interval between successive estimations. It uses a constant time interval (50 ms), instead of precise time measurements. In order to see how close it is to the actual variations, we measured time interval between successive executions. Note that the interval is hardly constant (unlike inter-arrival time) due to constraint on partition-budget as well as delay by higher-priority partitions/tasks. This can be seen from Figure 21 that compares the localization task's execution intervals under `P-RES` and `P-RES-NI`. Although the interval ranges between 30 and 60 ms with a few outliers, the average matches the task's estimation rate (20 Hz). With Blinder enabled, we measured the error in the position estimation (from the ground-truth) and observed a 3% increase in the error when compared to the case when time interval is measured from the wall clock.

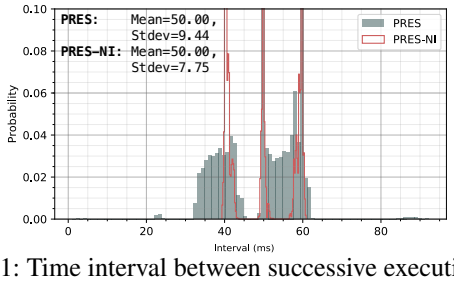


Figure 21: Time interval between successive execution of the localization task in Π_3.

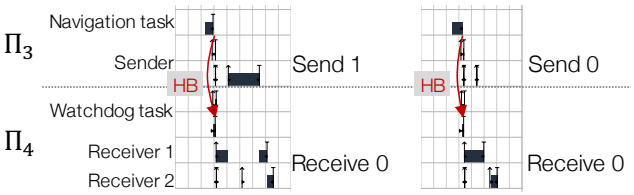


Figure 22: LITMUS^{RT} schedule traces of the covert channel scenario when Blinder is used.

Covert channel: Figure 22 shows the schedule traces for the scenario presented in Section 3.1 with Blinder enabled. The navigation task's heartbeat message helps the watchdog task in the health monitoring partition to time the receiver tasks to the sender. Nevertheless, due to Blinder, Receiver 1, which has a lower-priority than Receiver 2, always completes its execution before Receiver 2 increases the shared counter no matter how long the sender executes. Thus, the receiver's inference always results in '0' regardless of the sender's signal.

6.2 Cost of Blinder

Response time: To evaluate the cost of Blinder in a general setting, we measured task response times from a system that is comprised of various rate-groups. The partition and task parameters are shown in Table 1. We initially set both α and β to 1, which sets the system load to 80%. In order to add randomness to the executions and thus to create numerous variations in timings, task inter-arrival times are allowed to vary by 20 percent. Task priorities are assigned according to Rate Monotonic policy [27] (i.e., shorter period $\rightarrow$ higher priority). We used `rtspin` tool of LITMUS^{RT} to generate the real-time tasks. We compare our method, `P-RES-NI` (N), against `P-RES` (P) and `TDMA` (T). For `TDMA`, the major cycle is set to 50 ms.

Figure 23 summarizes the statistics of task response times obtained from running each scheme for 10 hours. The empirical probability distributions and the complete data including the analytic WCRTs can be found in Appendix B. The analytic WCRTs for `P-RES` are calculated by the analysis presented in [14]. Those under `TDMA` can be calculated similarly by treating other partitions as a single, highest-priority periodic task. The analyses assume no kernel cost, and thus the actual measurements can be slightly higher than what are numerically computed. The results highlight the following: (i) `P-RES-NI` does not increase the WCRTs compared to `P-RES`. This is because all the partitions are schedulable as discussed in Section 5.3; (ii) the behavior of Π_1 (i.e., the highest-priority partition) is not affected by Blinder because it never enters into the deferred release mode; (iii) the experimental WCRTs, in particular of low-priority tasks, under `P-RES-NI` tend to be smaller than those measured under `P-RES` especially in low-priority partitions. This is because those tasks are made more responsive by Blinder (i.e., deferred release of higher-priority tasks reduces the amount of preemption on lower-priority tasks), and thus the true worst-cases were less likely to be observed. In theory, the WCRTs under `P-RES-NI` and `P-RES` are same; (iv) while `TDMA` benefits low-priority partition in terms of the WCRTs due to guaranteed time slices, it degrades the average responsiveness; (v) due to the `lag`-based release control, the average-case response times, in particular of high-priority tasks, increase. The impact is more noticeable in lower-priority partitions. As a result, $\tau_{4,1}$'s average response time is increased by 20% under `P-RES-NI`. However, although criticality is not necessarily identical to priority, such low-priority partitions tend to be less sensitive to degraded responsiveness; (vi) the impact on lower-priority tasks are smaller. For instance, the average response times of $\tau_{3,4}$ and $\tau_{4,4}$ are decreased by 2% and 5%, respectively, compared to `P-RES`. This is because in addition to the reduced-preemptions by higher-priority tasks, a part or whole of the `lag` times could have been spent nevertheless to wait for higher-priority tasks to finish. Hence, their delayed releases can be masked.

Table 1: Partition (T_i, B_i) and task $(p_{i,j}, e_{i,j})$ parameters for the evaluation of response times on LITMUS^{RT} system. Units are in ms. The system load is controlled by α and β. For instance, the system load is 80% for $\alpha = \beta = 1$. $\texttt{Pri}(\Pi_i) > \texttt{Pri}(\Pi_{i+1})$.

	$\tau_{i,1}$	$\tau_{i,2}$	$\tau_{i,3}$	$\tau_{i,4}$
Π_1 $(20, 4\alpha)$	$(40, 2\beta)$	$(80, 4\beta)$	$(160, 8\beta)$	$(320, 16\beta)$
Π_2 $(30, 6\alpha)$	$(60, 3\beta)$	$(120, 6\beta)$	$(240, 12\beta)$	$(480, 24\beta)$
Π_3 $(40, 8\alpha)$	$(80, 4\beta)$	$(160, 8\beta)$	$(320, 16\beta)$	$(640, 32\beta)$
Π_4 $(50, 10\alpha)$	$(100, 5\beta)$	$(200, 10\beta)$	$(400, 20\beta)$	$(800, 40\beta)$

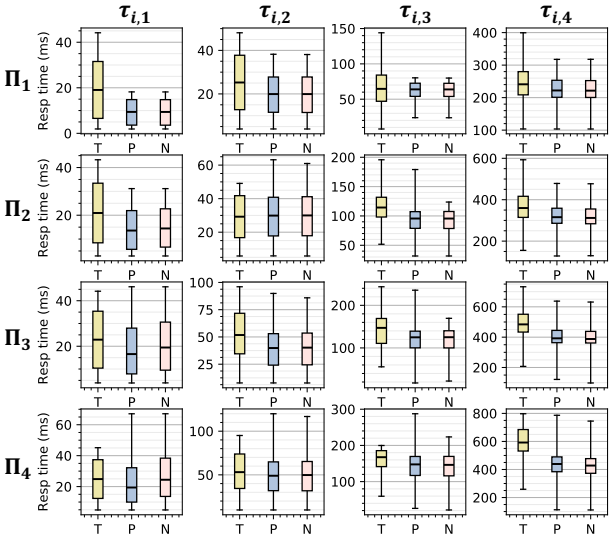


Figure 23: Statistics of response times of the tasks in Table 1.

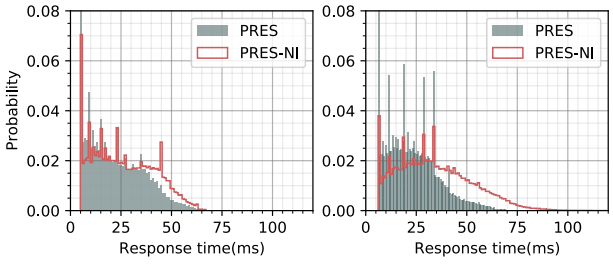


Figure 24: Probability distribution of $\tau_{4,1}$'s response times.

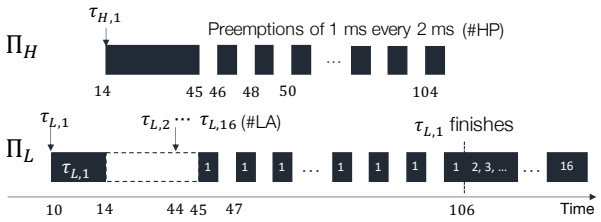


Figure 25: Configuration used for the overhead measurements with varying number of partition-level preemptions and size of arrival queue. $p_{*,*}$ =200 ms, T_*=200 ms, B_*=100 ms.

Table 2: Average and standard deviation of response times (ms) of $\tau_{L,1}$ in Figure 25.

#HP	12	18	24	30	30		
#LA	15				30	45	450
P-RES	96.15 (0.04)	96.17 (0.06)	96.19 (0.07)	96.22 (0.09)	96.22 (0.08)	96.23 (0.08)	96.39 (0.08)
P-RES-NI	96.17 (0.04)	96.19 (0.05)	96.22 (0.06)	96.24 (0.08)	96.25 (0.08)	96.26 (0.07)	96.73 (0.07)

Figure 24(b) shows the probability distribution of $\tau_{4,1}$'s response times when Π_4 is not schedulable due to the increased system load (by setting $\alpha = \beta = 1.25$ in Table 1). The complete measurements data are provided in Table 4 in Appendix B. Note that with this configuration, some tasks miss their deadlines even under P-RES. We performed this experiment to show the impact of Blinder on such an ill-configured system. Due to $\tau_{4,1}$'s longer release-delay, its worst and mean response times are increased by 41% and 45%, respectively.

Table 5 in Appendix B presents the worst-case and average-case response times when the system load is reduced to 40% (by setting $\alpha = \beta = 0.5$). The results do not show statistically significant evidence of a difference between P-RES and P-RES-NI, except that $\tau_{4,1}$'s average response time increases by 4.5% in P-RES-NI. Recall that the accuracy of the covert communication demonstrated in Section 3 increases significantly when the system is light-loaded. This suggests that Blinder can deter such malicious attempts effectively, incurring a small overhead on real-time applications.

Scheduling overheads: In our implementation of Blinder in LITMUS^{RT}, Linux high-resolution timer (hrtimer) is used to schedule the lag-based task release. When a partition is preempted, the timer is rescheduled upon resumption to account for the suspended time. Thus, the number of partition-level preemptions as well as the size of the arrival queue may affect the scheduling overhead. For this experiment, we use a two-partition configuration shown in Figure 25. Π_L enters into the deferred release mode at time 14 (ms) due to the preemption by Π_H. 15 tasks of Π_L arrive at time 44, right before Π_L returns from Π_H's preemption. Every 2 ms from time 46, Π_H preempts Π_L for 1 ms. Note that the result does not change with varying number of higher-priority partitions, because Blinder's complexity is independent of the number of partitions; it is irrelevant as to *who* preempts Π_L.

The initial lag values of the tasks that arrive at time 44 are 30. Hence, they are not released until $\tau_{L,1}$ executes for another 30 ms since returning from Π_H's preemption. We assigned the highest Π_L-local priority to $\tau_{L,1}$ to isolate any impact of task-level preemption. Therefore, the arrival queue of Π_L remains same until $\tau_{L,1}$ finishes at time 106. We vary the number of Π_H's preemptions, and accordingly the execution time of $\tau_{L,1}$ is adjusted to keep its nominal response time same. We ran each configuration for 300 seconds with and without Blinder. As Table 2 shows, $\tau_{L,1}$'s response time naturally increases with the number of Π_H's preemptions (denoted by #HP) in both cases. However, the difference of P-RES-NI from P-RES remains almost constant. Increasing the number of Π_L's tasks (denoted by #LA) that arrive at time 44, thus increasing the size of the arrival queue, does not change the trend. These results indicate that the overhead due to Blinder's lag-based task release is statistically insignificant and that it is scalable.

7 Related Work

Timing-channels are a major threat to information security [15, 18]. Microarchitectural timing-channels often involve shared hardware resources: cache [34], branch predictors [24], memory banks, TLBs, and interconnects [30]. An attacker usually either exploits the trace left by other users or overwhelms the bandwidth. Fuzzy-time [19, 42] introduces noise to system clocks so that adversaries cannot obtain precise physical time. Virtual time approaches [25, 45] enforce execution determinism by providing artificial time to processes. Although these techniques can mitigate timing-channels formed from observing physical time progress, they cannot prevent the scenarios presented in this paper because the tasks do not require any time information to perceive a change in other partition's temporal behavior.

Studies have shown that real-time scheduling can leak information, whether intended or not. Son et al. [38] provide a mathematical framework for analyzing the existence and deducibility of covert channels under Rate Monotonic scheduling. Similarly, Völp et al. [44] address the problem of information-flows that can be established by altering task execution behavior. The authors proposed modifications to the fixed-priority scheduler to close timing-channels while achieving real-time guarantees. In [43], Völp et al. also tackle the issues of information leakage through real-time locking protocols and proposed transformation for them to prevent unintended information leakage. All of these works address the problem of task-level information leakage, whereas our work concerns information-flow among time-partitions through varying partition-level behavior.

Formal verification can be used to prove absence of covert timing-channels. Murray et al. [31] show the non-interference of Sel4's time-partitions isolated by a static round-robin schedule. Liu et al. [29] prove that partitions can be dynamically scheduled, while preserving non-interference, if task arrivals are always aligned with partition's activation. Vassena et al. [41] present a language-level information-flow control

system to eliminate covert timing-channels by fixing the exact allocation of global-time for each parent-children thread group (analogous to the partition-task relationship), allowing user threads to access the global clock.

8 Conclusion

Blinder makes partition-local schedules deterministic by controlling the timing of task release and thus prevents local tasks from perceiving other partitions' varying behavior. We demonstrated that with Blinder, adversaries cannot form an algorithmic covert timing-channel through a hierarchical scheduling even if the system is configured in the most favorable way to them. Blinder enables applications to enjoy the level of flexibility that dynamic partitioning schemes would achieve while guaranteeing the partition obliviousness that static approaches would provide. Also, it is backward-compatible and minimally-intrusive in that no modification is required to the underlying (both global and local) scheduling mechanisms while incurring statistically insignificant overheads on the scheduler. Therefore, existing systems can benefit from the improved security and resource efficiency that it provides without a complete re-engineering, which is advantageous especially to safety-critical systems that require high re-certification costs.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by NSF grants 1945541, 1763399, 1715154, and 1521523. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

References

[1] LynxSecure. https://www.lynx.com/products/lynxsecure-separation-kernel-hypervisor.

[2] QNX Adaptive Partitioning Thread Scheduler. https://www.qnx.com/developers/docs/7.0.0/index.html#com.qnx.doc.neutrino.sys_arch/topic/adaptive.html.

[3] QNX Hypervisor. https://blackberry.qnx.com/en/software-solutions/embedded-software/industrial/qnx-hypervisor.

[4] QNX Platform for Digital Cockpits. https://blackberry.qnx.com/content/dam/qnx/products/bts-digital-cockpits-product-brief.pdf.

[5] Wind River Helix Virtualization Platform. https://www.windriver.com/products/helix-platform/.

[6] Wind River VxWorks 653 Platform. https://www.windriver.com/products/vxworks/certification-profiles/#vxworks_653.

[7] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, 1998.

[8] Aeronautical Radio, Inc. *Avionics Application Software Standard Interface: ARINC Specification 653P1-3*, 2010.

[9] Jim Alves-Foss, Paul W Oman, Carol Taylor, and Scott Harrison. The mils architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2(3/4):239–247, 2006.

[10] D. Bernstein. Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[11] Ma Bo, Mu Dejun, Fan Wei, and Hu Wei. Improvements the Seccomp Sandbox Based on PBE Theory. In *Proc. of the 27th Conference on Advanced Information Networking and Applications Workshops*, 2013.

[12] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, 2006.

[13] Matt Davis. Creating a vDSO: The Colonel's Other Chicken. *Linux J.*, 2011(211), 2011.

[14] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proc. of the 24th IEEE Real-Time Systems Symposium*, 2005.

[15] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *Proc. of the 14th EuroSys Conference*, 2019.

[16] Thomas Gleixner and Douglas Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Proc. of the Linux symposium*, volume 1, 2006.

[17] Gernot Heiser. The seL4 microkernel – an introduction (white paper). Revision 1.2. June 2020.

[18] Gernot Heiser, Gerwin Klein, and Toby Murray. Can we prove time protection? In *Proc. of the Workshop on Hot Topics in Operating Systems*, 2019.

[19] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[20] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *Proc. of 36th IEEE Symposium on Security and Privacy*, 2015.

[21] Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[22] Jung-Eun Kim, Tarek Abdelzaher, and Lui Sha. Budgeted generalized rate monotonic analysis for the partitioned, yet globally scheduled uniprocessor model. In

Proc. of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium, 2015.

[23] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *Proc. of the 51st IEEE/ACM International Symposium on Microarchitecture*, 2018.

[24] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proc. of 40th IEEE Symposium on Security and Privacy*, 2019.

[25] Peng Li, Debin Gao, and Michael K Reiter. Stopwatch: a cloud architecture for timing channel mitigation. *ACM Transactions on Information and System Security (TISSEC)*, 17(2):1–28, 2014.

[26] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proc. of 27th USENIX Security Symposium*, 2018.

[27] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[28] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[29] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. Virtual timeline: A formal abstraction for verifying preemptive schedulers with temporal isolation. *Proc. ACM Program. Lang.*, 4, December 2019.

[30] Yangdi Lyu and Prabhat Mishra. A Survey of Side-Channel Attacks on Caches and Countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, 2018.

[31] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In *Proc. of the 34th IEEE Symposium on Security and Privacy*, 2013.

[32] John Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory, 1992.

[33] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. *NASA Langley Technical Report*, Mar. 1999.

[34] S. Saxena, G. Sanyal, and Manu. Cache based side channel attack: A survey. In *Proc. of the International Conference on Advances in Computing, Communication Control and Networking*, 2018.

[35] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004.

[36] Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russell B Kegley, Dennis R Perlman, Greg Arundale, et al. Real-time computing on multicore processors. *Computer*, 49(9):69–77, 2016.

[37] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proc. of the 24th IEEE Real-Time Systems Symposium*, 2003.

[38] Joon Son and J. Alves-Foss. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems. In *Proc. of the IEEE Information Assurance Workshop*, 2006.

[39] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1:27–60, 1989.

[40] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, January 1995.

[41] Marco Vassena, Gary Soeller, Peter Amidon, Matthew Chan, John Renner, and Deian Stefan. Foundations for parallel information flow control runtime systems. In *Proc. of Principles of Security and Trust*, 2019.

[42] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *Proc. of the 3rd ACM Workshop on Cloud Computing Security*, 2011.

[43] Marcus Völp, Benjamin Engel, Claude-Joachim Hamann, and Hermann Härtig. On confidentiality preserving real-time locking protocols. In *Proc. of the 19th IEEE Real-Time Embedded Technology and Applications Symposium*, 2013.

[44] Marcus Völp, Claude-Joachim Hamann, and Hermann Härtig. Avoiding Timing Channels in Fixed-priority Schedulers. In *Proc. of the ACM Symposium on Information, Computer and Communications Security*, 2008.

[45] Weiyi Wu, Ennan Zhai, David Isaac Wolinsky, Bryan Ford, Liang Gu, and Daniel Jackowitz. Warding off timing attacks in deterland. In *Proc. of the Conference on Timely Results in Operating Systems*, 2015.

Appendix A Computation of $B_i(\mathtt{t_a}(\tau_{i,x}))$ in `ShiftRelease`

Suppose `ShiftRelease` occurs at time t, and let $\tau_{i,x}$ be the earliest-arrival task in the arrival queue. As shown in Algorithm 7, its arrival time, i.e., $\mathtt{t_a}(\tau_{i,x})$, becomes the beginning of a new deferred release mode. Here, we compute a new value

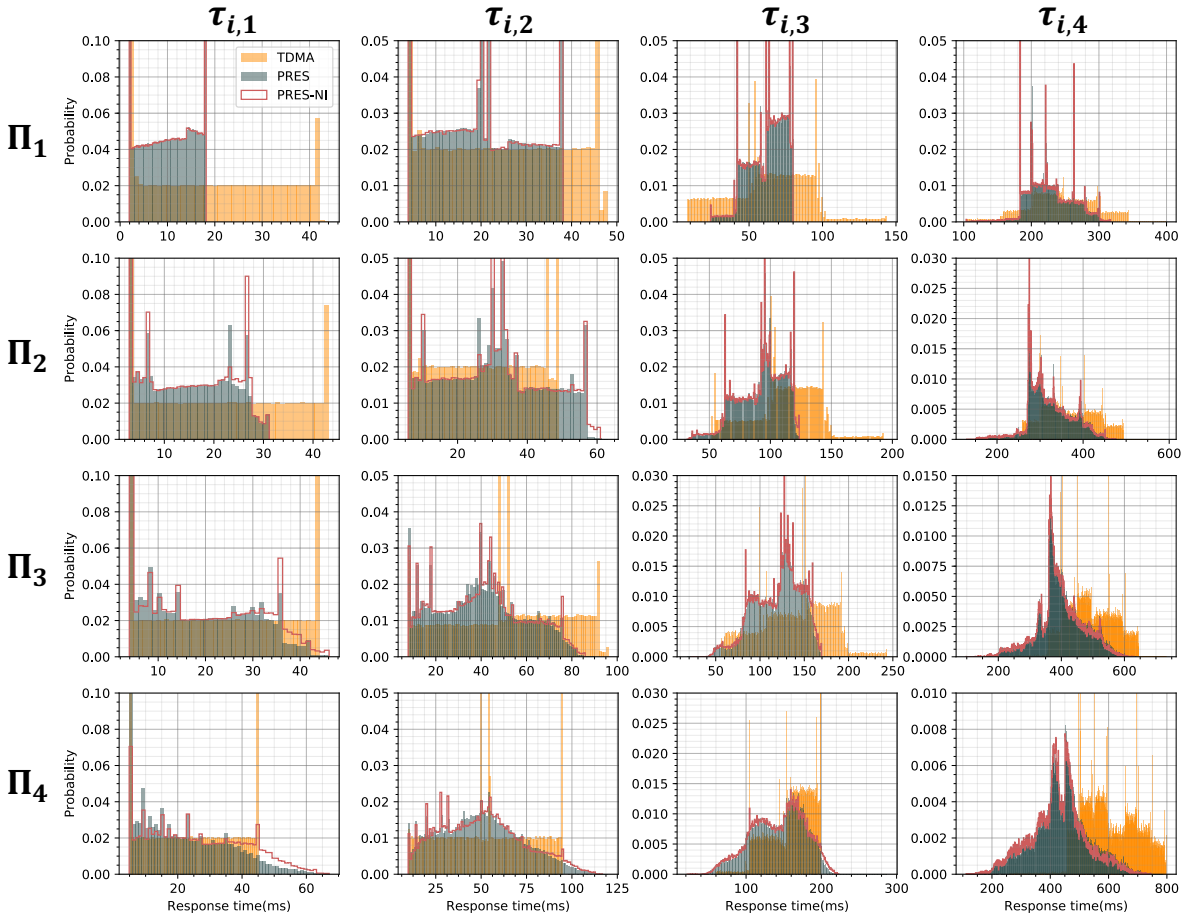


Figure 26: Empirical probability distributions of task response times under TDMA, P-RES, and P-RES-NI when $\alpha = \beta = 1$.

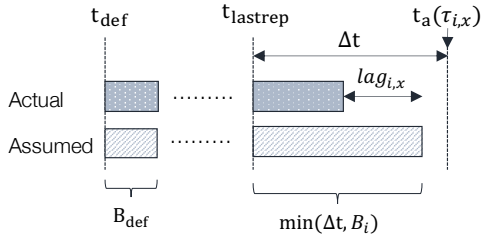


Figure 27: $t_a(\tau_{i,x})$ becomes the beginning of a new deferred release mode, and $B_i(t_a(\tau_{i,x}))$ is recomputed using $lag_{i,x}$.

for B_{def} based on $lag_{i,x}$ that is the remaining lag of $\tau_{i,x}$ when the ShiftRelease is happening. ShiftRelease occurs because $lag_{i,x}$ is a non-zero. That is, the executions released before $\tau_{i,x}$'s arrival (at time $t_a(\tau_{i,x})$) is shorter than what is assumed when calculating the initial lag of $\tau_{i,x}$. Hence, from $lag_{i,x}$ we can find how much budget of Π_i would have left for $\tau_{i,x}$ at its arrival.

We can consider two cases: there had been at least one budget replenishment before $t_a(\tau_{i,x})$ or not since entering the current deferred release mode (i.e., t_{def}). Figure 27 shows the former case. In this case, $t_{lastrep}$ is the time instant at which the last budget replenishment happens before $t_a(\tau_{i,x})$. At this moment, the budget is fully replenished to B_i. Now, $B_i(t_a(\tau_{i,x}))$ depends on how much of the budget is consumed by tasks of Π_i until $t_a(\tau_{i,x})$. If no tasks execute since $t_{lastrep}$, the full budget B_i would have been available at $t_a(\tau_{i,x})$. In this case, $lag_{i,x}$ is longer than $\min(t_a(\tau_{i,x}) - t_{lastrep}, B_i)$, that is, the maximum available time during the interval of $[t_{lastrep}, t_a(\tau_{i,x}))$. As $lag_{i,x}$ decreases, less budget becomes available. Therefore,

$$B_i(t_a(\tau_{i,x})) = B_i - \left[\min\left(t_a(\tau_{i,x}) - t_{lastrep}, B_i\right) - lag_{i,x}\right]_0,$$

where $[x]_0 = \max(x, 0)$. If there had been no budget replenishment since t_{def}, B_i and $t_{lastrep}$ are replaced by B_{def} and t_{def}, respectively.

Appendix B Complete Measurement Data of Response Time

Figure 26 shows the empirical probability distributions of task response times when the system shown in Table 1 (with $\alpha = \beta = 1$) in Section 6.2 is scheduled by TDMA, P-RES, and P-RES-NI. Tables 3, 4, and 5 show (i) the analytic worst-case response times (calculated by using the analysis in [14]) and (ii) experimental worst- and average-case response times.

Table 3: Experimental worst- and average-case response times (in ms) when $\alpha = \beta = 1$.

	Analytic WCRT		**TDMA** (T)			**P-RES** (P)			**P-RES-NI** (N)			**Δ Average**	
	TDMA	P-RES	Worst	Average	Stdev	Worst	Average	Stdev	Worst	Average	Stdev	(T-P)/P	(N-P)/P
$\tau_{1,1}$	42.00	18.00	44.15	19.65	13.46	18.19	9.47	5.74	18.18	9.45	5.73	107.50%	-0.21%
$\tau_{1,2}$	48.00	38.00	48.09	25.24	13.95	38.22	19.92	10.11	38.09	19.89	10.13	26.71%	-0.15%
$\tau_{1,3}$	144.00	80.00	144.01	63.57	26.83	80.23	62.71	12.14	79.90	62.59	12.17	1.37%	-0.19%
$\tau_{1,4}$	400.00	320.00	399.45	242.98	49.84	317.63	226.38	34.51	317.82	225.80	34.33	7.33%	-0.26%
$\tau_{2,1}$	43.00	31.00	43.26	21.33	13.57	31.16	14.02	8.61	31.16	14.69	8.66	52.14%	4.78%
$\tau_{2,2}$	49.00	64.00	49.09	28.92	13.85	63.18	29.85	14.71	60.96	30.15	14.87	-3.12%	1.01%
$\tau_{2,3}$	196.00	184.00	195.73	112.03	27.00	179.06	92.53	18.46	123.67	92.50	18.84	21.07%	-0.03%
$\tau_{2,4}$	600.00	664.00	592.93	364.63	64.89	478.56	323.98	51.16	477.04	320.96	51.13	12.55%	-0.93%
$\tau_{3,1}$	44.00	46.00	44.17	23.09	13.64	46.13	18.15	11.26	46.11	20.23	11.67	27.22%	11.46%
$\tau_{3,2}$	96.00	90.00	96.02	52.37	23.70	89.90	39.69	19.06	85.89	40.35	19.30	31.95%	1.66%
$\tau_{3,3}$	248.00	250.00	243.58	139.79	38.79	235.55	120.07	25.86	169.50	120.33	26.39	16.42%	0.22%
$\tau_{3,4}$	800.00	890.00	732.36	491.47	75.26	637.61	400.21	73.04	631.58	395.00	71.92	22.80%	-1.30%
$\tau_{4,1}$	45.00	67.00	45.15	24.87	13.67	67.08	21.96	13.69	67.10	26.37	14.89	13.25%	20.08%
$\tau_{4,2}$	95.00	128.00	95.06	53.82	24.29	119.97	49.58	22.44	116.81	50.31	22.95	8.55%	1.47%
$\tau_{4,3}$	200.00	328.00	199.78	161.71	29.21	287.48	142.65	34.67	223.48	142.54	35.45	13.36%	-0.08%
$\tau_{4,4}$	800.00	1128.00	797.48	607.31	92.55	786.56	438.05	94.71	745.06	427.14	92.11	38.64%	-2.49%

Table 4: Experimental worst- and average-case response times (in ms) when $\alpha = \beta = 1.25$.

	Analytic WCRT		**TDMA** (T)			**P-RES** (P)			**P-RES-NI** (N)			**Δ Average**	
	TDMA	P-RES	Worst	Average	Stdev	Worst	Average	Stdev	Worst	Average	Stdev	(T-P)/P	(N-P)/P
$\tau_{1,1}$	40.00	17.50	40.20	18.39	12.95	17.67	9.46	5.55	17.68	9.45	5.55	94.40%	-0.11%
$\tau_{1,2}$	47.50	37.50	47.55	25.16	13.65	37.65	20.52	9.63	37.58	20.49	9.64	22.61%	-0.15%
$\tau_{1,3}$	142.50	80.00	142.37	64.15	25.54	80.03	63.15	11.85	79.85	63.06	11.87	1.58%	-0.14%
$\tau_{1,4}$	400.00	320.00	398.42	239.97	48.22	316.44	227.36	34.05	316.15	227.18	34.11	5.55%	-0.08%
$\tau_{2,1}$	41.25	31.25	41.41	20.51	13.13	31.39	13.99	8.34	31.40	14.95	8.42	46.60%	6.86%
$\tau_{2,2}$	48.75	65.00	48.77	29.80	13.53	61.26	30.82	13.78	61.16	31.18	14.04	-3.31%	1.17%
$\tau_{2,3}$	195.00	185.00	194.31	111.08	26.48	179.58	92.71	18.18	124.54	92.97	18.64	19.81%	0.28%
$\tau_{2,4}$	600.00	665.00	591.01	365.91	63.16	477.13	321.29	50.97	474.49	317.92	50.95	13.89%	-1.05%
$\tau_{3,1}$	42.50	47.50	42.65	22.66	13.23	47.61	17.83	10.38	47.60	21.00	11.29	27.09%	17.78%
$\tau_{3,2}$	95.00	97.50	94.95	52.11	22.40	95.36	41.25	17.29	87.33	42.02	17.71	26.33%	1.87%
$\tau_{3,3}$	247.50	257.50	241.86	140.15	37.12	236.91	118.71	26.52	176.51	119.21	26.70	18.06%	0.42%
$\tau_{3,4}$	800.00	897.50	723.41	488.99	73.51	631.79	379.96	73.63	615.82	372.57	73.07	28.70%	-1.94%
$\tau_{4,1}$	43.75	93.75	43.86	24.82	13.26	88.13	23.45	12.79	124.45	34.08	18.17	5.84%	45.33%
$\tau_{4,2}$	93.75	162.50	93.72	55.01	23.11	156.96	56.45	21.73	167.54	62.28	24.24	-2.55%	10.33%
$\tau_{4,3}$	200.00	362.50	199.52	162.74	28.88	352.83	151.24	45.33	293.57	151.00	41.55	7.60%	-0.16%
$\tau_{4,4}$	800.00	1162.50	796.62	606.38	91.23	848.32	446.81	105.42	781.24	424.71	99.57	35.71%	-4.95%

Table 5: Experimental worst- and average-case response times (in ms) when $\alpha = \beta = 0.625$.

	Analytic WCRT		**TDMA** (T)			**P-RES** (P)			**P-RES-NI** (N)			**Δ Average**	
	TDMA	P-RES	Worst	Average	Stdev	Worst	Average	Stdev	Worst	Average	Stdev	(T-P)/P	(N-P)/P
$\tau_{1,1}$	46.00	19.00	47.18	22.28	14.12	19.20	9.47	6.01	19.21	9.45	6.01	135.27%	-0.21%
$\tau_{1,2}$	49.00	39.00	49.17	25.18	14.32	39.27	18.55	11.04	39.17	18.57	11.01	35.74%	0.11%
$\tau_{1,3}$	147.00	80.00	147.22	78.85	21.99	80.36	61.44	12.77	80.08	61.35	12.81	28.34%	-0.15%
$\tau_{1,4}$	400.00	320.00	639.17	259.47	42.66	319.41	223.67	35.18	318.73	222.62	35.15	16.01%	-0.47%
$\tau_{2,1}$	46.50	30.50	46.71	23.06	14.20	30.68	14.10	9.03	30.69	14.29	9.03	63.55%	1.35%
$\tau_{2,2}$	49.50	62.00	49.66	26.99	14.28	60.59	27.80	16.39	60.54	27.83	16.47	-2.91%	0.11%
$\tau_{2,3}$	198.00	182.00	197.99	113.49	27.82	144.64	91.76	19.37	121.92	91.61	19.48	23.68%	-0.16%
$\tau_{2,4}$	600.00	662.00	598.90	366.12	68.12	480.02	326.37	52.30	477.95	325.10	52.24	12.18%	-0.39%
$\tau_{3,1}$	47.00	43.00	47.21	24.00	14.22	43.09	18.58	12.03	43.08	19.18	12.10	29.17%	3.23%
$\tau_{3,2}$	98.00	85.00	98.19	52.99	26.30	83.08	36.85	21.66	83.00	36.99	21.75	43.80%	0.38%
$\tau_{3,3}$	249.00	245.00	247.12	142.19	40.96	211.52	121.09	26.20	164.84	120.98	26.52	17.43%	-0.09%
$\tau_{3,4}$	800.00	885.00	786.50	499.35	79.08	636.94	424.84	70.98	638.34	421.88	70.23	17.54%	-0.70%
$\tau_{4,1}$	47.50	56.50	47.69	24.93	14.23	56.65	23.12	15.11	56.58	24.17	15.23	7.83%	4.54%
$\tau_{4,2}$	97.50	109.00	97.71	54.80	26.32	106.56	45.97	26.92	106.45	46.03	27.07	19.21%	0.13%
$\tau_{4,3}$	200.00	309.00	294.37	159.48	29.94	294.48	149.41	33.47	208.70	148.94	34.21	6.74%	-0.31%
$\tau_{4,4}$	800.00	1109.00	799.10	607.05	94.49	790.65	510.74	89.55	794.26	507.11	89.77	18.86%	-0.71%