TimeDice: Schedulability-Preserving Priority Inversion for Mitigating Covert Timing Channels Between Real-time Partitions

Man-Ki Yoon
Yale University, USA

Jung-Eun Kim
Syracuse University, USA

Richard Bradford
Collins Aerospace, USA

Zhong Shao
Yale University, USA

***Abstract*—Timing predictability is a precondition for successful communication over a covert timing channel. Real-time systems are particularly vulnerable to timing channels because real-time applications can easily have temporal locality due to limited uncertainty in schedules. In this paper, we show that real-time applications can create hidden information flow even when the temporal isolation among the time partitions is strictly enforced. We then introduce an online algorithm that randomizes time-partition schedules to reduce the temporal locality, while guaranteeing the schedulability of, and thus the temporal isolation among, time partitions. We also present an analysis of the cost of the randomization on the responsiveness of real-time tasks. From an implementation on a Linux-based real-time operating system, we validate the analysis and evaluate the scheduling overhead as well as the impact on an experimental real-time system.**

***Keywords*-covert channel; timing channel; real-time systems;**

I. Introduction

Time is a crucial resource for enabling safety-critical applications to operate, monitor, and recover correctly. Especially when a system is integrated from applications with different levels of criticality, temporal isolation among them must be enforced to prevent faulty or malicious applications from misusing the CPU time resource. *Hierarchical scheduling* [1], [2] has been the key mechanism in high-assurance systems as a general approach to partitioning CPU time among real-time applications. It enforces time constraints on each application, which can use its share freely to run its local tasks. Hence, it can abstract away the details of how others use the assigned time resource, enabling modular reasoning about individuals' temporal behavior. Thus it has been successfully employed especially in avionics systems [3], [4] and is also increasingly adopted in other time-critical systems such as automotive systems [5], [6].

However, time is a powerful medium of *hidden* communication, especially in real-time systems because of their timing-predictable nature in operation [7]–[9]. In particular, sharing time among real-time components makes it possible for them to communicate indirectly by altering the way they consume time. In this paper, we demonstrate such an *algorithmic timing channel* [10], [11] between real-time partitions that are under strong *budget constraints* on CPU time and thus completely isolated from each other by a hierarchical scheduler. The technique builds a probabilistic model of the receiver's responsiveness; the sender modulates how it consumes its own budget, which influences the receiver's timing of CPU usage. A Bayesian inference enables the receiver to profile and predict the sender's signals even in the presence of noises due to other non-colluding time partitions. We also present a learning-based approach that finds patterns in the execution timings. We discuss conditions under which a system becomes more vulnerable to the threats.

We then introduce a partition-schedule randomization protocol, TimeDice, which is the main contribution of this paper. It reduces the temporal locality in partition schedules under a priority-based hierarchical scheduling by taking a rather radical approach: randomly *inverting* the priority relations among partitions. This adds noise to the execution timing, not to the time source [12], [13]. Hence, it is effective for systems in which it is difficult to completely remove every precise time source including external sources such as network services. Furthermore, it does not require modifications at the local scheduling level (i.e., within partition) [11].

The critical challenge in the partition-schedule randomization is that unprincipled randomization may lead partitions to *miss deadlines* – i.e., not being able to fully utilize the CPU budget assigned to it. Hence, at each scheduling decision point, TimeDice determines, on the fly, which partitions are allowed to take the CPU while not leading other partitions to under-use their budgets. Therefore, by construction, TimeDice guarantees a set of partitions to be *schedulable* if they were so before any randomization. We also show that a slight *bias* in the random selection in fact further reduces the level of temporal locality, thus making the covert timing channel more inaccurate. We analyze the impact of our partition-level randomization on the responsiveness of real-time tasks. From an implementation on a Linux-based real-time operating system, we evaluate the solution's impact on the scheduling overhead and task responsiveness as well as its effectiveness against the covert channel.

II. System Model

We consider N real-time partitions $\mathbf{\Pi} = \{\Pi_1, \ldots, \Pi_N\}$ that share the CPU time. The partitions are scheduled in a *hierarchical* manner as shown in Fig. 1; when a scheduling decision is to be made, a partition is selected first by the *global* scheduler. Then, the selected partition schedules its tasks according to its *local* scheduling policy. This paper considers a priority-based global scheduling, which is known to achieve improved responsiveness and CPU utilization compared to static-partitioning schemes [11], [14]. In this scheme, each partition is associated with a unique priority $\texttt{Pri}(\Pi_i)$ and the global scheduler selects the highest-priority partition. Real-time server algorithms, such as periodic server [15] and sporadic server [16], can instantiate a priority-based partition. Due to the enhanced CPU utilization, commercial RTOSes and real-time hypervisors [2], [6], [17] as well as open-source ones [18], [19] are increasingly supporting priority-based partition scheduling.

Explicit inter-partition communications through *overt* channels are handled by an OS-layer service (e.g., message-passing) which does not require synchronization between partitions. We

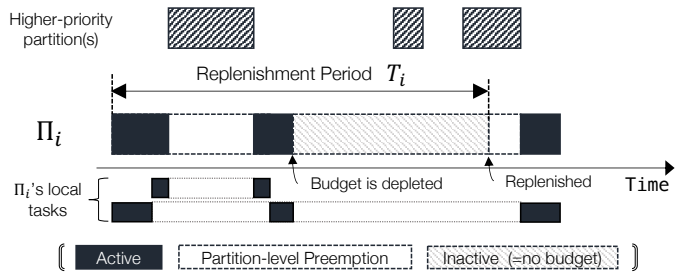


Fig. 1: Hierarchical scheduling. The local tasks of partition Π_i can run only when it has a budget.

do not consider shared-resource-based inter-partition communication that may require a synchronization protocol such as Priority Inheritance/Ceiling Protocols [20].

a) **Real-time partition and task models:** Each partition is associated with a *maximum budget* B_i and a *replenishment period* T_i; the partition can serve up to B_i (e.g., 10 ms) to its tasks during each period T_i (e.g., 100 ms) as shown in Fig. 1. We denote the *remaining* budget for time t by $B_i(t)$ and $0 \leq B_i(t) \leq B_i$. When a task of partition Π_i executes, $B_i(t)$ is depleted for the amount of task execution. No task of Π_i can execute when $B_i(t) = 0$ unless there is a higher-priority partition that has an unused budget but no task to run; the budget may be used by Π_i to (i) prevent additional interference by the *deferred executions* [21], [22] and (ii) to improve responsiveness because the CPU would otherwise be idled anyway. Hence, the lower-priority partition may end up using more than its budget. Nevertheless, this does not change the *worst-case* behavior of the higher-priority partition.

Each partition Π_i is comprised of a set of tasks $\Pi_i = \{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,|\Pi_i|}\}$. Each task is characterized by $\tau_{i,j} := (p_{i,j}, e_{i,j})$, where $p_{i,j}$ is the minimum inter-arrival time (also called as period) and $e_{i,j}$ is the worst-case execution time (WCET). That is, for each arrival it may execute for an arbitrary amount of time upper-bounded by $e_{i,j}$.

b) **Terminology:** Although it is a task that arrives and executes on CPU, we will use these terms to describe a partition's state for ease of explanation. A partition is said to (i) *arrive* if its budget is being replenished and (ii) *execute* when it has taken the CPU and one of its tasks runs. Also, it is said to be *active* if its remaining budget is non-zero. Otherwise, it is said to be *inactive*. Lastly, to guarantee the temporal isolation among partitions, each needs to be *schedulable*:

Definition 1 (Schedulable partition)**.** *Partition* Π_i *is said to be* schedulable *if it is guaranteed to serve its tasks for its maximum budget* B_i *over every replenishment period* T_i*.*

III. Covert Timing Channel Between Partitions

In high-assurance systems, information flows must be explicit (e.g., 'authorized channels' in Multiple Independent Levels of Security (MILS) systems [23], [24]). That is, any communication between partitions must be known and configured a priori. However, as applications are increasingly developed/supplied by third-party vendors in various forms and updated frequently, it is challenging to trust or verify them thoroughly. A hidden information flow, i.e., covert channel, can thus be used to leak

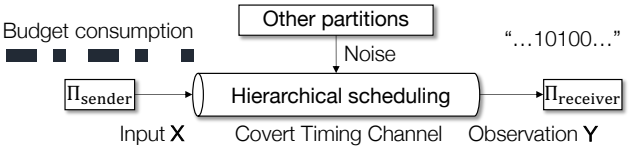


Fig. 2: Covert timing channel over hierarchical scheduling.

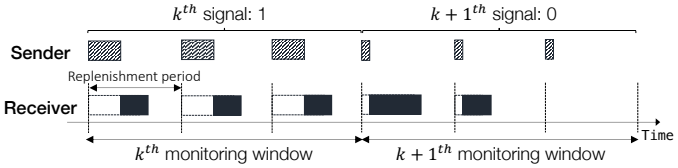
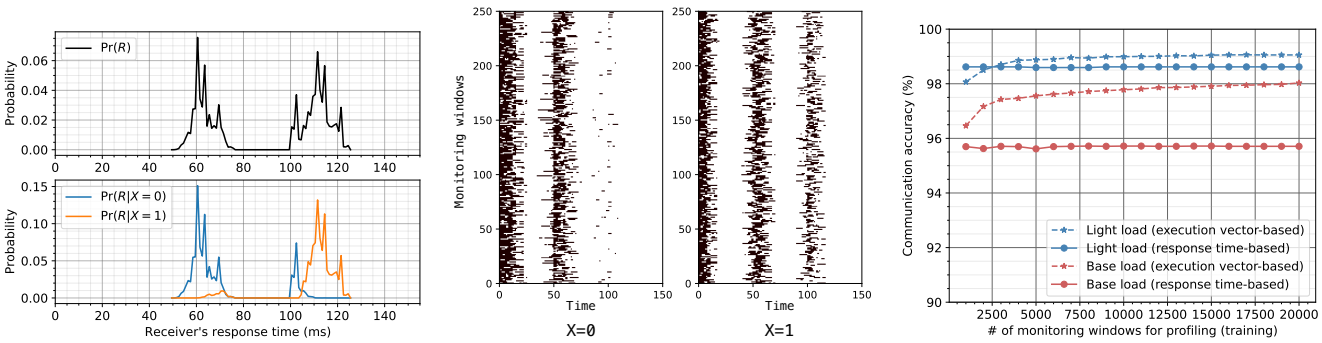


Fig. 3: The receiver observes how its execution timing changes to predict the sender's signal.

out sensitive information to a partition that is not allowed to obtain it through authorized channels. In this section, we show how hierarchical scheduling can be exploited by real-time partitions to create such a hidden information flow. As shown in Fig. 2, a sender partition tries to transmit a binary signal (0 or 1) by modulating the way it consumes its budget depending on each signal. The sender's varying behavior affects how/when the receiver partition consumes its budget. The receiver decodes its *local observation* to infer the sender's signal. The presence of other partitions, which share the CPU time with the sender and receiver, induces errors in the receiver's interpretation.

a) **Strategy and pre-conditions:** Fig. 3 illustrates a strategy for forming a covert timing channel between real-time partitions. The sender and receiver partitions have an agreed-upon time at which they start the *profiling* phase, during which the sender sends bits 0 and 1 alternately. When the sender wants to signal bit 1, it uses up its budget. Otherwise, it consumes its budget as little as possible. Meanwhile, the receiver observes how its local tasks execute. Unlike the mechanism used in [11] that requires coordination between two local tasks (as shown in Fig. 18 in Sec. V-C), a single task of the receiver partition measures times it takes to execute a block of code, i.e., *response times* (from arrival to finish). Hence, if it observes a relatively long response time, the sender partition has likely consumed its budget and thus has signaled bit 1. They may even form a multi-bit channel by dividing the response time range into multiple levels. Furthermore, the receiver could instead collect richer information about its execution (e.g., Fig. 4(b)) and apply a machine learning method, as will be explained shortly.

Although it is easy to coordinate the start time, their execution frequencies should be chosen in consideration of the budget constraints. Suppose that the replenishment periods of the sender and receiver partitions are T_S and T_R, respectively, where $T_S < T_R$. First, they need to agree on the length of the receiver's monitoring window, during which a single observation is made. Second, the *slower* replenishment rate determines the rate of their executions. In the example, the sender can execute as fast as the receiver because the latter's replenishment rate is slower. Once these are known, the sender can determine how many times it needs to execute during a monitoring window, which is three times in the example shown in Fig. 3.



(a) Probability distribution of the receiver's response times. The receiver profiles $\Pr(R|X=0)$ and $\Pr(R|X=1)$.

(b) Heatmap of execution vectors when the sender signals $X = 0$ or 1. Black: executed, white: not executed.

(c) Communication accuracy of covert channel between Π_2 and Π_4 in a 5-partition system. 10,000 samples are used for testing.

Fig. 4: (a) Response time-based and (b) learning-based profilings and (c) feasibility test.

b) **Profiling phase:** The presence of other partitions (other than the sender and receiver) in the system, however, can make the interpretation of a receiver's observation ambiguous; a long response time could be due to those other partitions preempting the receiver partition directly. Hence, a profiling phase is necessary. One way is to build probabilistic models $\Pr(R|X=0)$ and $\Pr(R|X=1)$, where R denotes the response time and X denotes the sender's signal which is unknown to the receiver. These probability distributions are estimated in the profiling phase by the receiver from a collection of measurements. Specifically, (i) the sender sends 0 and 1 alternately during the profiling phase and (ii) the receiver collects m measurements and divide them into two groups, $\mathcal{R}_{odd} = \{r_1, r_3, \ldots, r_{m-1}\}$ and $\mathcal{R}_{even} = \{r_2, r_4, \ldots, r_m\}$. Since the response time is likely to be shorter when the sender's signal is 0, the receiver estimates $\Pr(R|X=0)$ (resp. $\Pr(R|X=1)$) from the group whose mean value is smaller (resp. larger). Fig. 4(a) shows the probability distribution of the receiver's response times measured in an example system. The bottom plot shows the profiles $\Pr(R|X=0)$ and $\Pr(R|X=1)$ that the receiver learns during the profiling phase. Although they look separated, the bottom plot suggests that when a response time is, for instance, $R = 102$ ms, it is more likely that the sender has signaled bit 0 (the long response time is actually due to a large delay by those other than the sender), which cannot be explained by $\Pr(R)$ (shown in the top plot) alone.

c) **Communication phase:** In the communication phase, the receiver's goal is to infer the most likely signal X from each new observation, R. For example, the receiver can perform Bayesian inference: $\Pr(X|R) = \frac{\Pr(R|X)\cdot\Pr(X)}{\Pr(R|X=0)+\Pr(R|X=1)}$. Because the receiver does not know what message the sender is sending, it has no reason to believe that the sender treats bit 0 differently from bit 1, hence $\Pr(X=0) = \Pr(X=1)$. Then, the receiver can infer X by comparing the values for $\Pr(R|X=0)$ and $\Pr(R|X=1)$ given a measurement $R = r$: e.g., $X = 0$ if $\Pr(R=r|X=0) > \Pr(R=r|X=1)$.

d) **Learning-based approach:** As an alternative approach, we also present a learning-based scheme that learns patterns of *when* the receiver executes. For this, the receiver divides its monitoring window into M micro intervals and monitors if it was able to use the CPU during the i^{th} interval. Specifically, for each monitoring window, an *execution vector*, $v = (v_1, v_2, \ldots, v_M)$, is created, where each v_i is set to 1 if the receiver executed during the i^{th} interval, or 0 otherwise. The receiver collects a training set through the profiling phase. Each of the execution vectors in the training set is labeled as either 0 or 1 (i.e., sender's signal). Fig. 4(b) shows the 0/1 heatmap of execution vectors of length $M = 150$ collected over 500 monitoring windows. The receiver can apply a supervised learning method (e.g., Support Vector Machine, Random Forest) to train a model that can predict the sender's signal given a newly-observed execution vector in the communication phase.

e) **Motivating scenario:** To exemplify a scenario in which real-time partitions can communicate over the covert timing channel explained above, we deployed an implementation of the techniques on an experimental 1/10th-scale self-driving car platform composed of 4 partitions as shown in Fig. 5. We chose a Linux-based RTOS, LITMUS^{RT} [19], as the hierarchical scheduler because Linux-based RTOSes are increasingly adopted by high-end real-time systems. The local real-time tasks run as ROS (Robot Operating System) [25] nodes. Hence, explicit inter-partition communication is only allowed through the ROS' publish-subscribe channels over TCP/IP, which can easily be monitored. The vision-based steering control (Π_2) publishes a steering command, and the path planner (Π_3), which computes a series of waypoints from the current position to a destination, publishes a navigation command, both of which are subscribed by the top-level behavior controller (Π_1) to compute and send out a driving command to the actuators. These data are also sent to the logging partition (Π_4) for post-debugging. However, the precise location, which is a sensitive data item processed by the planner (Π_3), is not published to any partition.

Now, using the prototype system, we consider a scenario in which an ill-intentioned system operator collects the trace of the vehicle's precise location by leaking it out from the path planning partition (Π_3) to the logging partition (Π_4)

	Application	T_i	B_i
Π_1	Behavior control	10 ms	1 ms
Π_2	Vision-based steering	20 ms	10 ms
Π_3	Path planning	30 ms	3 ms
Π_4	Data logging	50 ms	5 ms

Fig. 5: Prototype 1/10th-scale self-driving car platform.

over a covert channel. For this, we applied the learning-based technique presented above. In particular, the planning task in Π_3 uses the period of 50 ms and modulates its execution length every three arrivals. At the receiver task in Π_4, we collected 3000 samples for training and evaluated its accuracy against 2000 test samples. Under this setting, the pair was able to achieve channel accuracy of 95.23%. However, it should be noted that an engineering effort would be required when applying these techniques to a full-scale system because it would create a higher level of channel noise. This demonstration highlights that systems that employ priority-based hierarchical scheduling can be vulnerable to covert timing channels.

f) **Feasibility test:** To show the feasibility of the scenarios presented above in a general setting, we run an example system of five synthetic partitions, $\mathbf{\Pi} = \{\Pi_i : 1 \leq i \leq 5\}$, on the same platform. The parameters are shown in Table I in Sec. V. The partitions are assigned different replenishment periods to represent various base-rate groups and also to remove potential bias due to the particular selection of the periods. Each partition is assigned a budget of size $B_i = 0.16T_i$, hence the total CPU utilization equals 80%. Π_2 is the sender and Π_4 is the receiver. The tasks in the other three partitions create unpredictable noise on the channel by varying their periods and execution times randomly (by up to 20%).

The receiver task measures its response time every 150 ms (i.e., $3 \cdot T_4$) and thus executes a code block that would take three full budget-replenishments of Π_4 in the worst-case. The sender partition consumes its budget as much (resp. little) as possible for three consecutive times to signal bit 1 (resp. 0). Fig. 4(c) shows that under this setting, the accuracy of the covert channel reaches about 95.7%. Also, such a high accuracy can be achieved without requiring a large number of samples for profiling. Higher accuracy can be achieved if the system is *lightly loaded* and thus creates less noise on the channel. We performed a similar experiment as the base setting above but with a half utilization: partition budgets and task execution times are cut by half. As shown in Fig. 4(c), the channel achieves 98.6% accuracy. In fact, as will be shown in later sections, our solution is more effective in such a scenario that is advantageous to the adversary.

We also evaluated the learning-based approach. Upon collecting a training set of execution vectors, the receiver applies the Support Vector Machine (SVM) [26] with Radial Basis Function kernel to train a classifier that can predict the sender's signal from an execution vector of the receiver. The learning-based approach achieves improved accuracy for both configurations, as shown in Fig. 4(c) ($\star$ markers). This is because an execution vector embeds more information than a response time; in fact, the latter can be derived from the former.

g) **Adversary model:** We assume any partition can be malicious and able to control the timing of its local tasks precisely. For instance, tasks can be launched at precise times using facilities originally intended to manage precedence constraints among tasks and to align 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. In addition, we do not address *microarchitectural* timing channels [27]–[29]. The algorithmic timing channel addressed in this paper can exist even if the microarchitectural timing channels were completely removed.

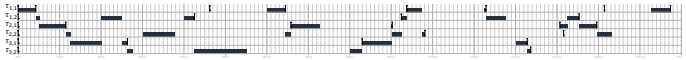
h) **Objective of the paper:** The covert timing channel presented above can be removed by a static time partitioning, such as the table-driven scheduling in IMA (Integrated Modular Avionics) architecture of ARINC 653 standard [1], because no two partitions can be active at any given instant. However, as studied in [11], [14], static partitioning schemes suffer from low CPU utilization. To remove the covert timing channel between *non-static* time partitions, (physical) time passage due to one's execution must not be observable by another. BLINDER [11] is based on a strong assumption that all sources of physical time are removed. However, in modern computer systems, many precise time sources are available, and hence it is difficult to eliminate every source of physical time. The goal of this paper is thus to *reduce* the capacity of the covert timing channel between real-time applications that (i) are dynamically partitioned and (ii) run in an environment where it is impossible to eliminate every source of physical time, (iii) while guaranteeing the real-time requirements (i.e., schedulability) of the partitions.

IV. SCHEDULABILITY-PRESERVING PARTITION SCHEDULE RANDOMIZATION

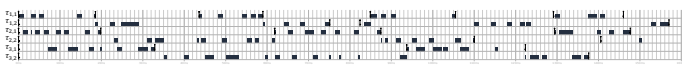
The covert timing channel presented in Sec. III is made possible by the priority relation between the sender and receiver partitions — the former, who has a higher priority, can affect the latter's execution in whatever way it wants. Hence, our solution, the TIMEDICE algorithm, is to invert the relationship *randomly* on the fly while preserving the partition-level schedulability; partitions are schedulable (Definition 1) if they were so before any randomization. Fig. 6 shows actual schedule traces for an example configuration when (a) partitions are scheduled by a fixed-priority policy (i.e., no randomization) and (b) they are randomized by TIMEDICE during run-time.

A. TIMEDICE *Algorithm*

The TIMEDICE algorithm picks a partition in a *non-deterministic* way instead of selecting the highest-priority partition. This *priority inversion*, however, could make some partitions unschedulable (i.e., missing their deadlines, thus underusing their budgets) if the candidates are chosen in an unprincipled way. Nonetheless, partition schedule should not be conservatively randomized. Hence, the key challenge lies in determining (a) which partitions are allowed to execute, and (b) how long a priority inversion is allowed. The TIMEDICE algorithm, shown in Algorithm 1, consists of two steps: (i) *candidate search*, which forms a list of candidate partitions



(a) Without TIMEDICE.



(b) With TIMEDICE.

Fig. 6: Actual schedule trace for a 3-partition example.

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

$\mathcal{L}_t = (\Pi_{(1)}, \Pi_{(2)}, \ldots, \Pi_{(n)}, \Pi_{IDLE})$ // active partitions

Step 1 – Candidate Search

$L_C \leftarrow \{\Pi_{(1)}\}$ // Candidate list

for $\Pi_{(i)} = \Pi_{(2)}, \ldots, \Pi_{(n)}, \Pi_{IDLE}$ **do**

  if CandidacyTest($\Pi_{(i)}, t$) == $False$ **then**

    Break // No need to test for $\Pi_{(i+1)}, \cdots$

  end

  $L_C \leftarrow L_C \cup \{\Pi_{(i)}\}$

end

Step 2 – Random Selection

$\Pi_x \leftarrow$ Select one from L_C

return Π_x

Algorithm 2: CandidacyTest($\Pi_{(i)}, t$)

for $\Pi_h \in hp(\Pi_{(i)}) - hp(\Pi_{(i-1)})$ **do**

  if SchedulabilityTest(Π_h, t) == $False$ **then**

    return $False$

  end

end

return $True$ // All $hp(\Pi_{(i)})$ are schedulable

allowed to take the CPU, and (ii) *random selection*, which selects one randomly from the list.

1) **Step 1 – Candidate search:** Algorithms 1 and 2 summarize the candidate search process. Suppose we are to make a scheduling decision at time t. Let $\mathcal{L}_t = (\Pi_{(1)}, \Pi_{(2)}, \ldots, \Pi_{(n)})$ be the list of active partitions, sorted in decreasing order of priority. Then, for each $\Pi_{(i)}$, starting from the highest priority, TIMEDICE tests if it can be a candidate. It passes the candidacy test if its execution at time t would still allow to meet deadlines for *all* of the higher-priority partitions, $hp(\Pi_{(i)})$, including partitions that are *not* active at present. If any of them would miss its deadline, $\Pi_{(i)}$ cannot be added to the candidate list, and the search process stops for time t. This is because if a higher-priority partition Π_h would miss its deadline due to the execution of $\Pi_{(i)}$, Π_h would miss the deadline due to $\Pi_{(i+1)}$ anyway. Note that the highest-priority active partition, $\Pi_{(1)}$, is always a candidate because no priority inversion occurs due to its execution. Meanwhile, if all of the active partitions pass the candidacy test, an additional test is performed to check if the CPU can be *idled*. This can be implemented by adding an 'imaginary' idle partition, Π_{IDLE}, and treating it as if it is another active partition, as shown in Algorithm 1. If passed, it is added to the candidate list.

a) **Detailed process of candidate search:** Suppose we are testing if $\Pi_{(i)}$ can be a candidate for time t. In order to check for this, the schedulability test (Algorithm 3) is performed against each $\Pi_h \in hp(\Pi_{(i)})$. Π_h is schedulable if and only if its *worst-case busy interval* does not end past its deadline. The busy interval begins with a priority inversion by $\Pi_{(i)}$ at time t:

Definition 2. *The level-Π_h busy interval with base time t and initial window of size w, denoted by $W_{h,t}(w)$ (shown in Fig. 7), is a time window $[t, t+q)$ that is comprised of the following:*

(a) *a priority inversion of size w by $\Pi_{(i)}$ during $[t, t+w)$,*

(b) *all remaining execution budgets of $hp(\Pi_h)$ as of time t,*

(c) *all the future executions of $hp(\Pi_h)$ that will arrive on or after t and are used up by the end of the busy interval,*

(d) *the remaining execution budget of Π_h itself.*

$W_{h,t}(w) = q = (a) + (b) + (c) + (d)$ is the length of the busy interval, and the end of the interval, i.e., $t+q$, is the first time instant when Π_h itself and all of $hp(\Pi_h)$ that arrive during $[t, t+q)$ use up their budgets if $[t, t+w)$ is taken by a low-priority partition $\Pi_{(i)}$.

Algorithm 3: SchedulabilityTest(Π_h, t)

$w \leftarrow$ MIN_INV_SIZE // length of priority-inversion

$W^0 \leftarrow w + B_h(t) + \sum_{\Pi_j \in hp(\Pi_h)} B_j(t)$ // Eq. (2)

$deadline \leftarrow r_{h,t} + T_h$ // $r_{h,t} + 2T_h$ if Π_h is inactive

while $W^{k+1}! = W^k$ **do**

  $W^{k+1} \leftarrow W^0 + \sum_{\Pi_j \in hp(\Pi_h)} \lceil (W^k(w) - o_{j,t})/T_j \rceil_0 \cdot B_j$

  if Π_h *is inactive* **then**

    $W^{k+1} \leftarrow W^{k+1} + \lceil (W^k(w) - o_{h,t})/T_h \rceil_0 \cdot B_h$

  end

  if $W^{k+1} > deadline$ **then**

    return *False* // Potential deadline miss

  end

end

return *True* // Π_h is schedulable

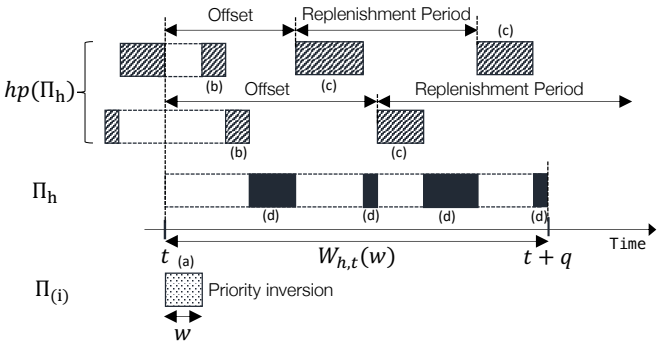


Fig. 7: Busy interval $W_{h,t}(w)$ is extended as long as Π_h is delayed by $hp(\Pi_h)$. $W_{h,t}(w) = (a) + (b) + (c) + (d)$.

Informally speaking, the level-Π_h busy interval represents how long it would take, in the worst-case, for Π_h to use up its remaining budget if it allows for a priority inversion of length w from time t to $t+w$. Finding $W_{h,t}(w)$ can be viewed as a *simulation* of the worst-case schedule from time t with a priority inversion of size w.

$W_{h,t}(w)$ for given t and w is computed iteratively as shown in Algorithm 3. Note that at time t, the amount of remaining budgets of all partitions (i.e., (b) and (d) in Fig. 7) are known. The *worst-case* busy interval is when all the *future* executions of $hp(\Pi_h)$ (i.e., (c) in Fig. 7) arrive as frequently as possible. This happens if they use up their budgets as soon as they become available. Since the last replenishment time of each $\Pi_j \in hp(\Pi_h)$ before t, denoted by $r_{j,t}$, is already known at time t, the relative *offsets* of their next replenishment time from t are known and calculated by $o_{j,t} = r_{j,t} + T_j - t$. Then, $W_{h,t}(w)$ is computed by the following iterative equation (similar to the approach to finding response time [30]):

$$W_{h,t}^{k+1}(w) = W_{h,t}^{0}(w) + \sum_{\Pi_j \in hp(\Pi_h)} \left\lceil \frac{W_{h,t}^{k}(w) - o_{j,t}}{T_j} \right\rceil_0 B_j, \quad (1)$$

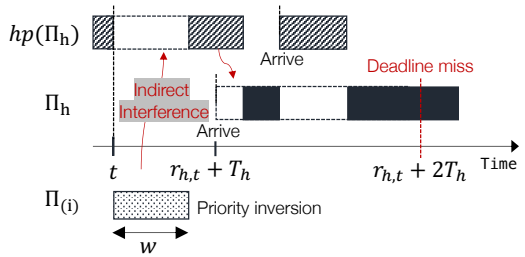


Fig. 8: A priority inversion by $\Pi_{(i)}$ at t can indirectly interfere with partition Π_h that will arrive *in the future* (at $r_{h,t}+T_h$).

where $\lceil x \rceil_0 = \max(\lceil x \rceil, 0)$. Here, the summand represents the interference from the higher-priority partitions (i.e., (c) in Fig. 7). $W^0_{h,t}(w)$, i.e., the initial busy interval, is comprised of (a), (b), and (d) in Fig. 7:

$$W^0_{h,t}(w) = w + B_h(t) + \sum_{\Pi_j \in hp(\Pi_h)} B_j(t). \tag{2}$$

Using the iterative procedure, $W_{h,t}(w)$ is computed as follows:

$$W_{h,t}(w) = \begin{cases} W^{k+1}_{h,t}(w) = W^k_{h,t}(w) & \text{if converging for some } k \\ \infty & \text{if not converging} \end{cases}.$$

It is the worst-case (i.e., longest possible) level-Π_h busy interval that starts with an execution of size w at t by a lower-priority partition. The scheduler tests if Π_h would still meet its deadline with the priority inversion in addition to the maximum interference from $hp(\Pi_h)$ by checking if the worst-case busy interval ends by its deadline, i.e., the next replenishment time:

$$t + W_{h,t}(w) \le r_{h,t} + T_h. \tag{3}$$

Note that the busy-interval analysis presented above assumes no synchronization between partitions (e.g., for overt inter-partition communication as explained in Sec. II).

b) **Indirect interference on inactive Π_h:** The analysis above assumes that Π_h is active at time t. One may overlook the case when Π_h is *inactive*, concluding that a priority inversion at present would not affect the future execution of Π_h; thus, no schedulability test for the inactive Π_h is needed. However, the scheduler also needs to test if the *upcoming* execution of Π_h, shown in Fig. 8, who is not active at present but would arrive at the next replenishment time $r_{h,t}+T_h$ at the earliest, would meet its upcoming deadline $r_{h,t}+2T_h$. The reason is that a priority inversion at time t by a lower-priority partition can *indirectly* interfere with the future execution of Π_h: higher-priority partition(s), i.e., $hp(\Pi_h)$, are delayed by the priority inversion from t to $t+w$, which creates a cascading delay that interferes with the execution of Π_h that will arrive in the future. Hence, the schedulability test should be performed against inactive partitions as well. For such a case, Eq. (1) can be simply extended, as shown in Algorithm 3, to include the upcoming execution of Π_h as another higher-priority partition (i.e., part of (c) in Fig. 7). If the busy interval ends before Π_h's upcoming arrival at $r_{h,t}+T_h$, the new term would be zero.

c) **Search complexity:** The schedulability test needs to be performed at most once for each partition in the system. Thus the search complexity is $\mathcal{O}(|\mathbf{\Pi}|)$. This can be explained better with an example shown in Fig. 9. Suppose there are 9

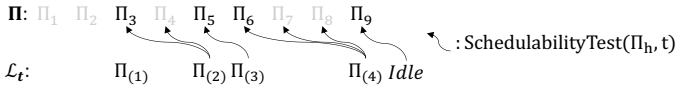


Fig. 9: For $\Pi_{(i)}$'s candidacy test, we only need to check those $\Pi_h \in hp(\Pi_{(i)})$ that have not been examined when testing $\Pi_{(i-1)}$.

partitions in the system, and $\mathcal{L}_t = (\Pi_3, \Pi_5, \Pi_6, \Pi_9)$ are active at present. First, $\Pi_{(1)} = \Pi_3$, and hence no schedulability test is performed because, as explained earlier, its execution does not make a priority inversion. Now, for $\Pi_{(2)} = \Pi_5$, we need to check if its execution for a length of w could make $\Pi_{(1)} = \Pi_3$ potentially unschedulable. We also need to check for Π_4, which is inactive, due to a possibility of the indirect interference explained above. For $\Pi_{(3)} = \Pi_6$, we do not need to test for Π_3 and Π_4 again because if they are schedulable in the presence of $\Pi_{(2)}$'s priority inversion, they are still schedulable when it is made by $\Pi_{(3)}$. That is, from their point of view, it does not matter who is creating a priority inversion, which is why the analyses in Eq. (1) and Eq. (2) depend only on the size of a priority inversion, w, not on *who* causes the priority inversion. Hence, for each $\Pi_{(i)}$ we only need to consider those $\Pi_h \in hp(\Pi_{(i)})$ that have not been examined when testing $\Pi_{(i-1)}$.

2) **Step 2 – Weighted random selection of partition:** Once a list of candidate partitions is found, the scheduler picks one randomly as shown in Algorithm 1. One may pick a partition uniform randomly: each candidate has an equal chance of $\frac{1}{|L_C|}$. Counter-intuitively, this can make a schedule *less randomized*. Consider an example in Fig. 10 in which a selection is being made between two partitions, $L_C = \{\Pi_1, \Pi_2\}$, at time t. Ignoring the idling option, the probability of Π_1 being selected for t is 1/2. If another selection is made at $t+1$, the probability is again 1/2. Hence, the probability that Π_1 would use up its budget during $[t, t+2)$ is 1/4, which is high considering the time until the deadline. Hence, it would likely finish too early.

In order to alleviate such biases, we propose a *weighted* selection process that considers the remaining budget and time until the deadline. Suppose a selection is made at time t. For each candidate $\Pi_i \in L_C$, the scheduler computes the *remaining utilization*:

$$u_{i,t} = B_i(t)/(d_{i,t} - t),$$

where $d_{i,t}$ is the deadline of Π_i as of time t (i.e., the next replenishment time), $d_{i,t} = r_{i,t} + T_i$, as in Eq. (3). Then, each candidate is assigned a normalized weight $\omega_{i,t} = u_{i,t} / \sum_{\Pi_x \in L_C} u_{x,t}$. If the CPU can be idled, the option is assigned a weight of $1 - \sum_{\Pi_x \in L_C} u_{x,t}$. Then, the scheduler performs a weighted

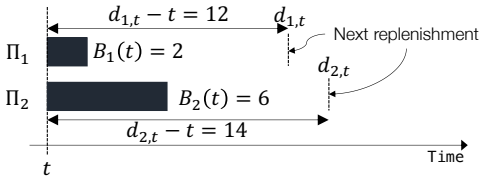


Fig. 10: With uniform random selection, Π_1 is likely to use up its budget *too early*. Hence, it should be given a smaller weight to decrease the chance of temporal locality.

random selection based on the $\omega_{i,t}$ values. This process can be viewed as a type of lottery scheduling [31] with the ticket allocation proportional to the remaining utilization.

Note that the weight reflects how urgent a partition is. That is, the weight is higher if the remaining budget is larger and/or the deadline is closer. The following theorem shows why assigning weights *inversely* proportional to the remaining utilization can actually increase the degree of temporal locality:

Theorem 1. *Giving a higher weight to a partition with a lower remaining utilization increases the degree of temporal locality.*

Proof: Let us consider a 2-candidate situation, $L_C = \{\Pi_j, \Pi_k\}$, for time t and assume that $u_{j,t} > u_{k,t}$. Suppose Π_k, who has a smaller remaining utilization, is selected and runs until $t+1$. Hence, $B_k(t+1) = B_k(t) - 1$. Then, the difference in the remaining utilization between the two *grows*:

$$u_{j,t+1} = \frac{B_j(t+1)}{d_{j,t} - t - 1} = \frac{B_j(t)}{d_{j,t} - t - 1} > \frac{B_j(t)}{d_{j,t} - t} = u_{j,t},$$

$$u_{k,t+1} = \frac{B_k(t+1)}{d_{k,t} - t - 1} = \frac{B_k(t) - 1}{d_{k,t} - t - 1} < \frac{B_k(t)}{d_{k,t} - t} = u_{k,t}.$$

Because $u_{j,t+1} - u_{k,t+1} > u_{j,t} - u_{k,t}$, the chance that Π_k would be selected again for time $t+1$ becomes larger. ■

Hence, higher weights should be given to those partitions with *larger* remaining utilization. This is because the weight of a partition decreases (resp. increases) if it is selected (resp. not selected), which steers the weights of candidate partitions in the direction towards being leveled as time proceeds. Therefore, one's budget consumption is likely to spread across a wide range, and accordingly, the chance of premature budget exhaustion, thus temporal locality, is reduced. As will be shown in Sec. V, the weighted random selection further increases the level of randomness in a partition schedule. The effect is more profound especially when the system is lightly loaded, which is when an adversary can achieve a higher communication accuracy, as discussed in Sec. III.

B. Schedulability Analysis

The schedulability of real-time tasks is tightly dependent on a particular choice of partition-local scheduling policy as well as budget replenishment policy. Hence, we base our analysis on the fixed-priority preemptive local task scheduling [32] on which our implementation is based. Let us first consider the case *without* TIMEDICE. The worst-case response time of each task can be computed by the analysis in [33]. In a nutshell, the worst-case situation for task $\tau_{i,j}$ of partition Π_i happens when (a) it arrives, with all the higher-priority tasks in the same partition, when Π_i's budget has been depleted as soon as possible; (b) their subsequent invocations arrive as frequently as possible; and (c) Π_i's budget supply is delayed as maximally as possible by higher-priority partitions. That is, the analysis finds how many budget replenishments are needed to serve the maximum workload from the local tasks.

Now, when partitions are scheduled by TIMEDICE, the worst-case is when the partition is maximally delayed in the last part towards the end of the period, as depicted in Fig. 11. It can be

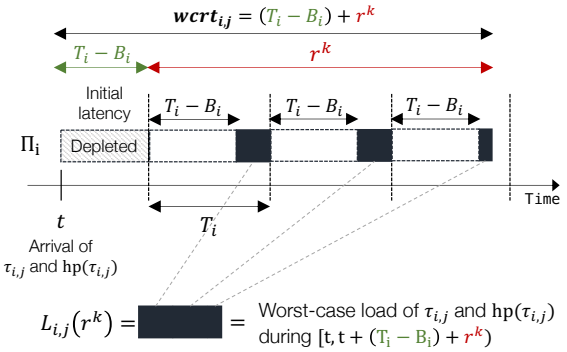


Fig. 11: The worst-case response time of $\tau_{i,j}$ when partition schedule is randomized by TIMEDICE.

computed by the following iterative procedure:

$$r^{k+1} = L_{i,j}(r^k) + \left\lceil \frac{L_{i,j}(r^k)}{B_i} \right\rceil (T_i - B_i), \tag{4}$$

where $L_{i,j}(r^k)$ is the worst-case task load demanded by task $\tau_{i,j}$ and those who have higher-priority than $\tau_{i,j}$ in the same partition from time t (when they arrive together) until $t + (T_i - B_i) + r^k$, and it is calculated as follows [33]:

$$L_{i,j}(r^k) = e_{i,j} + \sum_{\tau_{i,x} \in hp(\tau_{i,j})} \left\lceil \frac{(T_i - B_i) + r^k}{p_{i,x}} \right\rceil e_{i,x}, \tag{5}$$

where $e_{i,*}$ and $p_{i,*}$ are the worst-case execution time and the minimum inter-arrival time of task $\tau_{i,*}$, respectively. r^0 in Eq. (4) can be initialized to $e_{i,j}$. Simply speaking, Eq. (4) finds how many budget replenishments of partition Π_i are needed to serve the workload of amount $L_{i,j}(r^k)$. The worst-case response time (WCRT) of task $\tau_{i,j}$ is $wcrt_{i,j} = (T_i - B_i) + r^k$ when r^k converges, and the task is schedulable if and only if it is not greater than the deadline. Notice that the WCRT of a task depends only on the parameters of the partition that it belongs to. Thanks to this modularity, the partition developer can use the WCRT analysis presented here to test in advance whether the tasks will meet their deadlines when TIMEDICE is used.

V. EVALUATION

A. Implementation

TIMEDICE is implemented in the latest version of LITMUS^{RT} [19] with kernel version of 4.9.30. It is applied to the sporadic-polling server of LITMUS^{RT} which is a variant of the sporadic-server algorithm [16]. In fact, TIMEDICE can also be applied to other priority-based server algorithms such as periodic server [15] and deferrable server [34]. Meanwhile, TIMEDICE does not affect the local scheduling policy.

The default global scheduler of LITMUS^{RT} selects the highest-priority partition among the active ones at every scheduling decision, and it takes the CPU until the next scheduling point that occurs upon task completion/arrival or budget depletion. When TIMEDICE is enabled, a partition (which is randomly selected) can use the CPU for the *quantum* size (`MIN_INV_SIZE` in Algorithm 3, which is set to 1 ms in our implementation) unless a certain event (e.g., task completion/arrival, budget depletion) occurs before the end of the quantum. Hence, the randomization happens approximately every 1 ms.

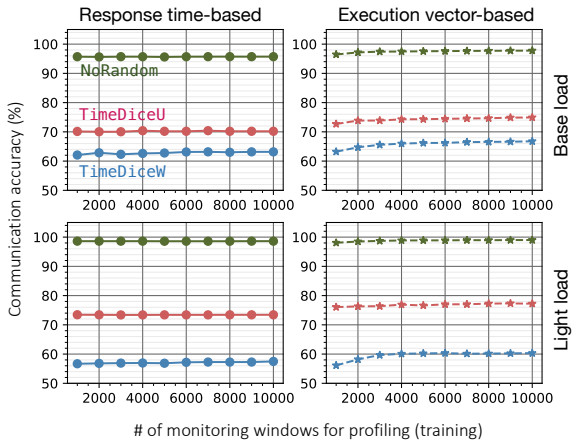


Fig. 12: Impact of TIMEDICE on the accuracy of communication over the covert timing channel. Base load: 80%. Light load: 40%. 10,000 samples are used for testing.

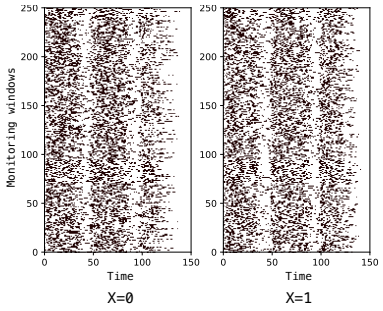


Fig. 13: Heatmap of execution vectors when partitions are randomized by TIMEDICE. Compare these with Fig. 4(b).

The implementation is deployed to the 1/10th-scale self-driving car platform explained in Sec. III. The system runs on Intel NUC mini PC [35] with Core i5-7260U processor operating at 2.20 GHz and a main memory of 8 GB.

B. Evaluation Results

We denote by `NoRandom` and `TimeDice` the default global scheduler of LITMUS^{RT} and TIMEDICE-enabled scheduler, respectively. Additionally, `TimeDiceW` and `TimeDiceU` are used to distinguish the weighted and uniform random selection of partition, respectively. By default, `TimeDice` indicates `TimeDiceW`.

1) **Covert-channel accuracy:** We extend the feasibility test presented in Sec. III by measuring the accuracy of communication over the covert channel when TIMEDICE is used. Fig. 12 shows the impact of TIMEDICE on channel accuracy. The x-axis is the number of monitoring windows used for profiling. The results highlight the following: (i) TIMEDICE is more effective when the system is more vulnerable to the covert channel, which can be observed from the 'Light load' case (bottom plots): `TimeDiceW` reduces the accuracy from 98.62% and 98.99% to 57.49% and 60.32% for the response time-based and execution vector-based approaches, respectively, indicating that the communication over the channel is not significantly better than a random guess (50%). This is mainly because partitions have more room to allow priority inversion when the system is lightly loaded, hence active partitions, including CPU idling, are more likely to pass the candidacy test (as

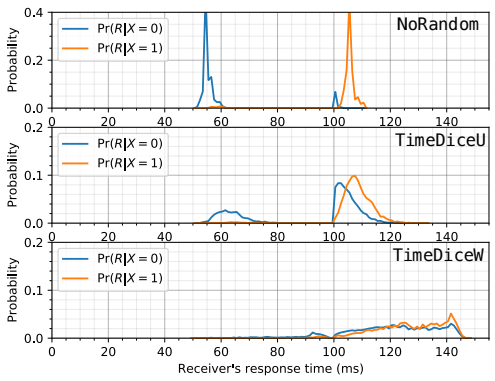


Fig. 14: Probability distribution of the receiver's response time in the light load configuration.

presented in Sec. IV-A1); (ii) Although `TimeDiceU` can already reduce the communication accuracy significantly, the weighted random selection further enhances the effectiveness of the randomization; (iii) the learning-based approach that uses execution vectors achieves higher accuracy than the response time-based approach even when partition schedule is randomized. This is because, as discussed in Sec. III, it contains richer information about the receiver's execution timing. However, TIMEDICE can still defend effectively against such a learning-based approach because with TIMEDICE, the receiver's execution is scattered across a wider range as shown in Fig. 13. Unlike the cases in Fig. 4(b), the sender's varying signal (i.e., left vs right) does not create distinctive patterns in the receiver's execution vectors.

The effect of TIMEDICE can be explained best by the probability distribution of the receiver's response times shown in Fig. 14. First of all, as can be seen from the middle plot, `TimeDiceU` makes $\Pr(R|X=0)$ and $\Pr(R|X=1)$ similar to each other, thus making it ambiguous to infer the most likely signal X given a new measurement R. However, the temporal locality, albeit reduced, still remains even with `TimeDiceU`. The weighted-random selection (`TimeDiceW`) spreads the receiver's execution range, thus finally little to no information can be gained from the probabilistic models.

Although TIMEDICE increases the uncertainty in the partition-level schedule by construction, we quantitatively evaluate its impact from an information-theoretic view. In particular, we measure the channel capacity [36], which is defined by $C = \max_{p(X)}(H(X) - H(X|R))$, where $p(X)$ is the input distribution, and $H(X)$, which is the entropy of channel input X, is maximized when $p(X)$ is a uniform distribution. It represents the *average reduction in uncertainty* about the channel input X after observing a response time R. Here, $H(X|R)$ is the channel noise and can be calculated by

$$H(X|R) = \sum_{R}\sum_{X} \Pr(X,R) \log \frac{\Pr(R)}{\Pr(X,R)}. \qquad (6)$$

Now, to measure the channel capacity, we consider a binary signal $X = 0, 1$ that follows a uniform distribution. Fig. 15 compares the channel capacity calculated from 10,000 samples. It is upper-bounded by $H(X) = 1$ as X is a binary signal following a uniform distribution, which is when the uncertainty about the input signal X is completely removed once observing

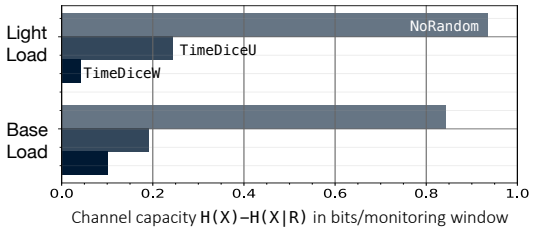


Fig. 15: Channel capacity in bits per monitoring window.

R, i.e., $H(X|R) = 0$. As the other extreme, it is lower-bounded by 0, which is achieved when $H(X) = H(X|R)$; that is, no information about the input X is conveyed by an observation R (e.g., when using a static partitioning such as TDMA). The unit is in the *bits per monitoring window*. The actual 'bits per second' depends on the execution frequency of the sender and the receiver. If the frequency of the monitoring window is f Hz and a single bit is transmitted per window, the results in Fig. 15 can be interpreted that about $0.8f$–$0.9f$ bits can be sent over 1 second under `NoRandom` and about $0.1f$–$0.2f$ bits per second under TIMEDICE. As can be seen from these results, TIMEDICE significantly reduces the channel capacity by introducing high noise $H(X|R)$ into the channel. This can be explained by $H(X|R) = \sum_{r \in R} \Pr(R = r) H(X|R = r)$ that rewrites Eq. (6); TIMEDICE makes (i) the set of possible response times, i.e., R, larger and (ii) the difference between $\Pr(X = 0|R = r)$ and $\Pr(X = 1|R = r)$ smaller for each particular r, i.e., $H(X|R = r)$ increases.

We also performed the same experiment on the 1/10th-scale self-driving car platform explained in Sec. III, but with TIMEDICE enabled. It was able to drop the accuracy of the covert channel between the path planning partition and the logging partition to 56.30% (from 95.23%).

2) **Task responsiveness:** We evaluate the cost of TIMEDICE on task responsiveness by measuring task response times from the 5-partition system shown in Table I. Task priorities follow Rate Monotonic policy [32], i.e., a task with a shorter period is assigned a higher priority, and we assume implicit deadline (=minimum inter-arrival time). The real-time tasks are generated by the `rtspin` tool of LITMUS^{RT}. We run the system for 10 hours for `NoRandom` and `TimeDice`, respectively.

The box plots in Fig. 16 show the spreads and centers of the response time measurements. We can first see that the range of response times is likely to extend with TIMEDICE, which indicates increased uncertainties in partition executions. The trend stands out more clearly in the high-priority partitions.

TABLE I: Partition replenishment period (T_i) and task's minimum inter-arrival time ($p_{i,j}$) for the evaluation of response times. Partition budget B_i and task's worst-case execution time $e_{i,j}$ are proportional to T_i and $p_{i,j}$, respectively: $B_i = \alpha T_i$, $e_{i,j} = \beta p_{i,j}$. By default, $\alpha = 16\%$ and $\beta = 3\%$. $\texttt{Pri}(\Pi_i) > \texttt{Pri}(\Pi_{i+1})$ and $\texttt{Pri}(\tau_{i,j}) > \texttt{Pri}(\tau_{i,j+1})$.

	$\tau_{i,1}$	$\tau_{i,2}$	$\tau_{i,3}$	$\tau_{i,4}$	$\tau_{i,5}$
Π_1 (20 ms)	40 ms	80 ms	160 ms	320 ms	640 ms
Π_2 (30 ms)	60 ms	120 ms	240 ms	480 ms	960 ms
Π_3 (40 ms)	80 ms	160 ms	320 ms	640 ms	1280 ms
Π_4 (50 ms)	100 ms	200 ms	400 ms	800 ms	1600 ms
Π_5 (60 ms)	120 ms	240 ms	480 ms	960 ms	1920 ms

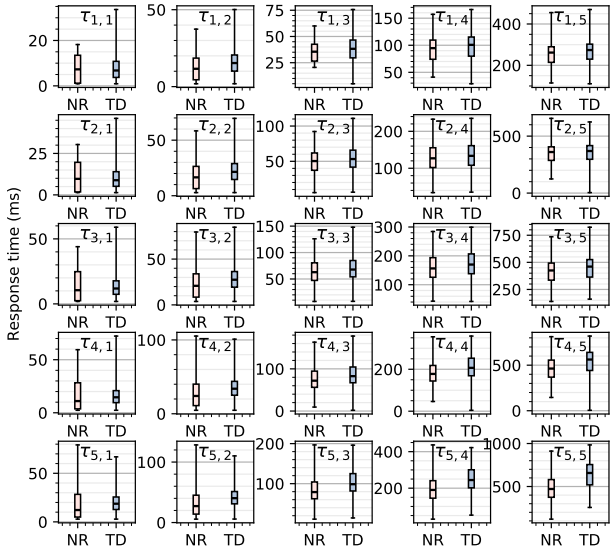


Fig. 16: Task response times when partitions are not randomized (`NR`) or randomized by TIMEDICE (`TD`).

TABLE II: Analytic and empirical worst-case response times (in ms). The tasks are schedulable because WCRT$\leq$ Deadline.

		`NoRandom(NR)`		`TimeDice(TD)`		`TD − NR`	
	Deadline	**Anal.**	**Empr.**	**Anal.**	**Empr.**	**Anal.**	**Empr.**
$\tau_{1,1}$	40.00	18.00	18.09	34.80	33.63	16.80	15.54
$\tau_{1,2}$	80.00	37.20	37.36	55.20	50.11	18.00	12.75
$\tau_{1,3}$	160.00	60.00	60.01	76.80	75.67	16.80	15.66
$\tau_{1,4}$	320.00	158.40	157.11	235.20	165.98	76.80	8.87
$\tau_{1,5}$	640.00	598.80	455.08	616.80	469.95	18.00	14.87
$\tau_{2,1}$	60.00	30.20	30.36	52.20	46.14	22.00	15.78
$\tau_{2,2}$	120.00	59.00	58.35	82.80	69.61	23.80	11.26
$\tau_{2,3}$	240.00	93.20	92.23	115.20	110.87	22.00	18.64
$\tau_{2,4}$	480.00	330.80	232.81	352.80	235.01	22.00	2.20
$\tau_{2,5}$	960.00	903.20	655.69	925.20	624.29	22.00	-31.40
$\tau_{3,1}$	80.00	44.00	43.98	69.60	58.96	25.60	14.98
$\tau_{3,2}$	160.00	84.80	79.55	110.40	84.77	25.60	5.22
$\tau_{3,3}$	320.00	128.00	126.01	153.60	147.90	25.60	21.89
$\tau_{3,4}$	640.00	444.80	284.35	470.40	299.19	25.60	14.84
$\tau_{3,5}$	1280.00	1208.00	735.69	1233.60	823.93	25.60	88.24
$\tau_{4,1}$	100.00	59.40	59.44	87.00	72.29	27.60	12.85
$\tau_{4,2}$	200.00	110.40	105.21	138.00	101.02	27.60	-4.19
$\tau_{4,3}$	400.00	167.60	163.24	192.00	177.71	24.40	14.47
$\tau_{4,4}$	800.00	560.40	354.82	588.00	358.25	27.60	3.43
$\tau_{4,5}$	1600.00	1517.60	812.75	1542.00	819.43	24.40	6.68
$\tau_{5,1}$	120.00	79.60	79.20	104.40	66.94	24.80	-12.26
$\tau_{5,2}$	240.00	145.60	128.33	165.60	110.18	20.00	-18.15
$\tau_{5,3}$	480.00	210.40	196.74	230.40	196.10	20.00	-0.64
$\tau_{5,4}$	960.00	685.60	436.94	705.60	422.19	20.00	-14.75
$\tau_{5,5}$	1920.00	1830.40	911.86	1850.40	983.67	20.00	71.81

Without any randomization, the high-priority partitions tend to experience little or no delay when they have budget and tasks to run. Hence, their tasks are likely to be served quickly. With the randomization by TIMEDICE, those partitions experience increased delays even when there are no other partitions, and so do their tasks. For the same reason, the average-case response times also increase in most cases, and the largest increase is 34.03% which is observed from $\tau_{5,2}$.

Table II compares the worst-case response times (WCRTs) that are analytically computed (columns labeled as **Anal.**) and experimentally measured (columns labeled as **Empr.**). The real-time requirement states that the WCRT of a task must

TABLE III: Impact of TIMEDICE on the responsiveness of the prototype self-driving applications. Units are in ms.

		NoRandom			TimeDice		
	Deadline	avg	std	max	avg	std	max
Behavior control	20	0.91	2.51	10.04	2.45	2.51	18.03
Vision-based steering	50	10.55	3.85	33.92	23.20	3.69	34.69
Path planning	50	0.62	0.79	6.35	1.06	2.29	19.83

TABLE IV: End-to-end latency of TIMEDICE's randomization.

Percentile	25%	50%	75%	99%	100%
$\|\mathbf{\Pi}\| = 5$	0.609 us	0.938 us	1.430 us	6.917 us	38.726 us
$\|\mathbf{\Pi}\| = 10$	1.156 us	2.079 us	3.266 us	20.500 us	54.915 us
$\|\mathbf{\Pi}\| = 20$	3.602 us	5.691 us	9.052 us	52.673 us	73.217 us

not exceed the deadline. The system designer can perform a schedulability test before deploying the system by calculating the analytic WCRT and checking if it meets the deadline. The analytic WCRTs for NoRandom cases are calculated by the analysis in [33] while those for TimeDice are calculated by the analysis presented in Sec. IV-B. Notice first that because the analyses assume zero kernel-overhead, the empirical WCRT can be slightly higher than what is numerically computed, albeit in rare cases (e.g., $\tau_{1,1}$). The results first highlight that all tasks are schedulable in both cases. Note, however, that this does not mean that TIMEDICE always preserves the *task* schedulability. Depending on partition and task configurations, some tasks may be unschedulable in the worst-case due to the additional delay in the randomized partition-level schedule. Nevertheless, in most cases, the difference in the analytic WCRT did not exceed one replenishment period of the partition that each task belongs to. As explained in Fig. 11 in Sec. IV-B, this follows the worst-case assumption that the last part of task execution is maximally delayed by $T_i - B_i$, whereas this delay can be as short as zero when partitions are not randomized. Hence, the difference is unlikely to exceed T_i as a rough bound unless a substantially large amount of load is added due to the extended busy interval.

Meanwhile, the empirical WCRTs of some tasks (e.g., $\tau_{2,5}$) are smaller with TIMEDICE than with NoRandom. This is simply because the true worst cases were not captured although we allowed tasks to vary the execution times and inter-arrival times for added variations and also ran the system for long hours. As analyzed earlier, tasks cannot have shorter WCRTs when partition schedule is randomized by TIMEDICE. If the system was run indefinitely, the difference in the empirical WCRTs (the last column in Table II) would have been non-negative.

Lastly, we evaluate the impact of TIMEDICE on the responsiveness of the tasks running on the prototype platform (Fig. 5). Note that the data logger is a collection of callback functions for logging data received from the others. Hence, we do not measure its response time. Similar to the benchmark results above, Table III shows that the average-case and (empirical) worst-case response times increase under TIMEDICE. Nevertheless, the tasks still meet their real-time requirements.

3) **Scheduling overhead:** TIMEDICE incurs overhead on the partition scheduler as it performs a candidate search with schedulability test at each scheduling decision. Hence, we measure the associated cost using the system in Table I. In addition to the 5-partition configuration, we double and quadruple the number of partitions by duplicating the partitions while adjusting the partition budgets and task execution times accordingly so that the total system utilization remains the same.

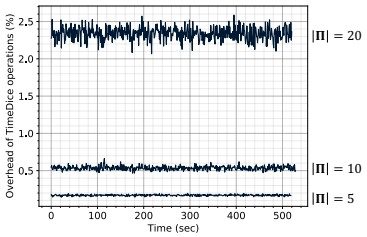


Fig. 17: Overhead of TIMEDICE operations (measured every second) for different number of partitions ($|\mathbf{\Pi}| = 5, 10, 20$).

TABLE V: Number of scheduling decisions and partition-switches for different number of partitions ($|\mathbf{\Pi}| = 5, 10, 20$).

	# Decisions/sec		# Switches/sec	
	NoRandom	TimeDice	NoRandom	TimeDice
$\|\mathbf{\Pi}\| = 5$	441.50	1333.69	247.55	911.86
$\|\mathbf{\Pi}\| = 10$	821.59	1725.93	467.85	1243.38
$\|\mathbf{\Pi}\| = 20$	1592.71	2594.09	907.59	1986.96

Table IV summarizes the end-to-end latency of Algorithm 1, that is, the time taken to pick a partition from the active ones. The results can be interpreted better by taking into account the frequency of scheduling decisions. Hence, we measured time spent by the TIMEDICE operations over every second. Fig. 17 shows that the scheduler spends about 1.7 ms in total over 1000 ms (thus overhead of 0.170%) on randomizing the schedules of 5 partitions. Similarly, the overhead for $|\mathbf{\Pi}| = 10$ and $|\mathbf{\Pi}| = 20$ are around 0.535% and 2.338%, respectively.

The randomization also causes more frequent scheduling decisions and partition switches than the no-randomization case. Table V shows the numbers of scheduling decisions and partition switches per unit time measured by running the 5, 10, 20-partition systems. One thing to notice from the results is that while the rate of decisions proportionally increases with the system size under NoRandom, it does not with TimeDice. This is due to the quantum-based randomization (i.e., MIN_INV_SIZE $=$ 1 ms in Algorithm 3). Recall that scheduling decisions are also made upon certain events such as task arrival/completion and budget depletion. In theory, TimeDice may demand 1000 additional scheduling points, which matches the trend shown in the results in Table V.

C. Comparison to BLINDER

BLINDER [11] cannot defend against the type of covert channel presented in this paper because it requires a rather strong assumption that every precise time source must be eliminated. Otherwise, as demonstrated in Sec. III, partitions can directly perceive (from the physical time) interference from other partitions. Even a networking operation (e.g., the data logging partition in Fig. 5 performing a remote logging) can serve as an external time source. Unlike BLINDER, TIMEDICE can be applied to such a system.

Although the result is self-evident, we implemented the BLINDER algorithm and performed the feasibility test in Sec. III again. For the base configuration and 10,000 samples for profil-

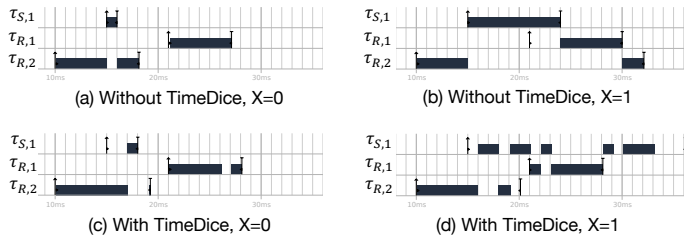


Fig. 18: The covert channel scenario from BLINDER [11]. The sender $\tau_{S,1}$ varies its length to signal $X = 0$ or 1. The receiver's inference is based on the order between $\tau_{R,1}$ and $\tau_{R,2}$.

ing, the accuracy is 95.67% and 97.73% for the response time-based and learning-based approaches, respectively, which are the same as what `NoRandom` achieved, indicating that BLINDER cannot defend against the covert channel presented in this paper.

Conversely, can TIMEDICE defend against the type of covert channel considered in [11]? The covert channel that BLINDER defends against uses the order between two local tasks, $\tau_{R,1}$ and $\tau_{R,2}$, in the receiver partition as shown in Fig. 18(a) and (b). The order between $\tau_{R,1}$ and $\tau_{R,2}$ is influenced by the length of the sender's preemption. With TIMEDICE, a long preemption by the sender (i.e., X=1) is likely to be split in a random manner as shown in Fig. 18(d); hence the receiver is likely to make a wrong prediction on the sender's signal.

Even if every source of precise time were eliminated, BLINDER requires each partition to faithfully implement the BLINDER's local-schedule transformation algorithm. Hence, partition-local schedulers should be trustworthy. Therefore, BLINDER cannot be applied to systems where local schedulers are not modifiable (e.g., partition supplied as a binary executable). On the other hand, TIMEDICE can be applied to such systems because it is a global-schedule transformation technique; hence, only the system integrator needs to be trustworthy.

As TIMEDICE allows the existence of physical time sources, it cannot reduce the channel capacity to zero (as shown in Fig. 15). This implies that communication over covert timing channel is still possible but at a slow rate. Hence, TIMEDICE is useful when the value of information leaked through a channel is transient – i.e., it diminishes faster than communication speed.

VI. RELATED WORK

Real-time hierarchical scheduling has been studied mostly by means of isolation mechanism for temporal reasoning, i.e., modular schedulability analysis [15], [33], [37], whereas little to no attention has been paid to the security implication of time-partitioning schemes. Yoon *et al.* [11] address an algorithmic timing channel through hierarchical scheduling that exploits changes in the order of partition-local tasks. As explained in detail in Sec. V-C, BLINDER can be used only when no precise time sources are available. Fuzzy-time [12], [13] and Virtual time [38], [39] make system clocks imprecise, which may degrade the usability of applications. In contrast, TIMEDICE adds noise to the execution timing, not to the time source.

Scheduler timing channels have been studied mainly at the task levels. Son *et al.* [40] showed that the rate monotonic scheduling is exposed to covert timing channel due to its scheduling timing constraints. Völp *et al.* [41] close such a channel by making the task executions deterministic: e.g., switching to an idle thread if a task stops early. Chen *et al.* [42] demonstrated a different type of threat against fixed-priority scheduling; an observer task infers the timings (e.g., future arrival times) of certain tasks by observing its own execution intervals. Such an attack is possible because of the timing determinism of real-time systems [7], [43]. However, such timing predictability can also help improve the security of real-time systems. For instance, one can fingerprint electronic control units using periodic Controller Area Network (CAN) messages to detect intrusion into in-vehicle network such as message replay and injection attacks [9], [44], [45].

Covert timing channels have been studied extensively in the network domain. A covert network timing channel leaks information by modulating intervals between packets [46], [47]. A straightforward solution is to control the network traffic by, for example, adding random delays to network packets [48], [49]. Randomization is in fact a critical ingredient for moving target defense (MTD) techniques [50], [51]. Davi *et al.* [52] used address space layout randomization (ASLR) [53] to randomize program code on the fly for each run to deter code-reuse attacks. Crane *et al.* [54] improved code randomization by enforcing execute-only memory to eliminate code leakage that allows an attacker to learn about the address space layout. Kc *et al.* [55] took a finer-grained approach that creates a process-specific instruction set that is hard to be inferred by an adversary. Zhang *et al.* [56] addressed the problem of information leakage through cache side-channels by randomly evicting cache lines and permuting memory-to-cache mappings. Jafarian *et al.* [57] considered MTD in software-defined networking (SDN), in which the controller randomly assigns (virtual) IP addresses to hosts in order to hinder adversaries from discovering targets.

VII. CONCLUSION

In this paper, we have demonstrated techniques that exploit a priority-based time-partitioning to create a covert timing channel between real-time partitions. As a solution, we have presented TIMEDICE, an online algorithm that reduces observable determinism in partition schedules by randomly allowing priority inversions while guaranteeing CPU budgets allocated to partitions. We have shown that TIMEDICE significantly raises the bar against the timing-based algorithmic covert channel and that it is more effective when the system is configured in a favorable way to an adversary. TIMEDICE will allow modern real-time systems to employ advanced functionalities enabled by a rich software ecosystem by increasing the level of security in the integration of real-time applications.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and the shepherd, Le Guan, for their valuable comments and suggestions. This work is supported in part by NSF grants 1945541 and 2019285, the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4018. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of their employers or sponsors.

REFERENCES

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

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

[3] "LynxOS-178," https://www.lynx.com/products/lynxos-178-do-178c-certified-posix-rtos.

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

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

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

[7] S. Vadineanu and M. Nasri, "Robust and accurate period inference using regression-based techniques," in *Proceedings of the IEEE Real-Time Systems Symposium*, 2020.

[8] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha, "TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2016.

[9] K.-T. Cho and K. G. Shin, "Fingerprinting electronic control units for vehicle intrusion detection," in *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.

[10] M. Vassena, G. Soeller, P. Amidon, M. Chan, J. Renner, and D. Stefan, "Foundations for parallel information flow control runtime systems," in *Proceedings of International Conference on Principles of Security and Trust*, 2019.

[11] M.-K. Yoon, M. Liu, H. Chen, J.-E. Kim, and Z. Shao, "Blinder: Partition-oblivious hierarchical scheduling," in *Proceedings of the 30th USENIX Security Symposium*, 2021.

[12] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of computer security*, vol. 1, no. 3-4, pp. 233–254, 1992.

[13] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in xen," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security*, 2011.

[14] J.-E. Kim, T. Abdelzaher, and L. Sha, "Budgeted generalized rate monotonic analysis for the partitioned, yet globally scheduled uniprocessor model," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015.

[15] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proceedings of the 24th IEEE Real-Time Systems Symposium*, 2003.

[16] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Journal of Real-Time Systems*, vol. 1, pp. 27–60, 1989.

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

[18] S. Xi, J. Wilson, C. Lu, and C. Gill, "Rt-xen: Towards real-time hypervisor scheduling in xen," in *Proceedings of the 9th ACM International Conference on Embedded Software*, 2011.

[19] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings of the 27th IEEE Real-Time Systems Symposium*, 2006.

[20] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[21] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Proceedings of the IEEE Real-Time Systems Symposium*, 1988.

[22] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292, 1993.

[23] Information Assurance Directorate (National Security Agency), "U.s. government protection profile for separation kernels in environments requiring high robustness," 2007.

[24] J. Alves-Foss, P. W. Oman, C. Taylor, and S. Harrison, "The mils architecture for high-assurance embedded systems." *International Journal of Embedded Systems*, vol. 2, no. 3/4, pp. 239–247, 2006.

[25] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[26] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, p. 273–297, Sep. 1995.

[27] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[28] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," in *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture*, 2018.

[29] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of 40th IEEE Symposium on Security and Privacy*, 2019.

[30] M. Joseph and P. K. Pandya, "Finding response times in a real-time system." *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[31] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible Proportional-Share resource management," in *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, 1994.

[32] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

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

[34] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 73–91, Jan. 1995.

[35] "Intel NUC Kit NUC7i5BNK," https://www.intel.com/content/www/us/en/products/boards-kits/nuc/kits/nuc7i5bnk.html.

[36] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[37] L. Almeida and P. Pedreiras, "Scheduling within temporal partitions: response-time analysis and server design," in *Proceedings of the 4th ACM International Conference on Embedded Software*, 2004.

[38] P. Li, D. Gao, and M. K. Reiter, "Stopwatch: a cloud architecture for timing channel mitigation," *ACM Transactions on Information and System Security (TISSEC)*, vol. 17, no. 2, pp. 1–28, 2014.

[39] W. Wu, E. Zhai, D. I. Wolinsky, B. Ford, L. Gu, and D. Jackowitz, "Warding off timing attacks in deterland," in *Proceedings of the Conference on Timely Results in Operating Systems*, 2015.

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

[41] M. Völp, C.-J. Hamann, and H. Härtig, "Avoiding Timing Channels in Fixed-priority Schedulers," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2008.

[42] C.-Y. Chen, S. Mohan, R. Pellizzoni, R. Bobba, and N. Kiyavash, "A novel side-channel in real-time scheduler," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2019.

[43] M. Salem, M. Crowley, and S. Fischmeister, "Anomaly detection using inter-arrival curves for real-time systems," in *Proceedings of the 28th Euromicro Conference on Real-Time Systems*, 2016.

[44] X. Ying, G. Bernieri, M. Conti, and R. Poovendran, "Tacan: Transmitter authentication through covert channels in controller area networks," in *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, 2019.

[45] C. Young, H. Olufowobi, G. Bloom, and J. Zambreno, "Automotive intrusion detection based on constant can message frequencies across vehicle driving modes," in *Proceedings of ACM Workshop on Automotive Cybersecurity*, 2019.

[46] S. Cabuk, C. E. Brodley, and C. Shields, "Ip covert timing channels: Design and detection," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[47] J. Xing, Q. Kang, and A. Chen, "Netwarden: Mitigating network covert channels while preserving performance," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[48] A. Belozubova, A. Epishkina, and K. Kogos, "Random delays to limit timing covert channel," in *Proceedings of European Intelligence and Security Informatics Conference*, 2016.

[49] Y. Wang, P. Chen, Y. Ge, B. Mao, and L. Xie, "Traffic controller: A practical approach to block network covert timing channel," in *Proceedings of the International Conference on Availability, Reliability and Security*, 2009.

[50] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving target defense: creating asymmetric uncertainty for cyber threats*. Springer Science & Business Media, 2011, vol. 54.

[51] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques," *IEEE Security & Privacy*, vol. 12, no. 2, pp. 16–26, 2014.

[52] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013.

[53] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[54] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[55] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[56] T. Zhang and R. B. Lee, "New models of cache architectures characterizing information leakage from cache side channels," in *Proceedings of the Annual Computer Security Applications Conference*, 2014.

[57] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: Transparent moving target defense using software defined networking," in *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networks*, 2012.