On Evading Randomization-Based Defense in Hierarchical Real-Time Systems

VIJAY BANERJEE, Washington State University, USA
SENA HOUNSINOU, Metro State University, USA
YANYAN ZHUANG, University of Colorado Colorado Springs, USA
MONOWAR HASAN, Washington State University, USA
GEDARE BLOOM, University of Colorado Colorado Springs, USA

Security for real-time systems is increasingly important with the growth of connected real-time systems in safety-critical domains such as automotive, medical, and avionics. A crucial aspect of securing such systems is to understand the attacks that the current techniques cannot effectively safeguard against. Especially relevant are vulnerabilities of real-time systems arising from their rigid temporal guarantees and attacks that exploit such vulnerabilities. Randomization-based defense techniques can reduce side-channel inference, but such techniques are limited due to the strict timing bounds of real-time systems. In this paper, we design and analyze NosyNeighbor, an inter-partition side-channel attack that exploits the timing guarantees of real-time systems to infer the timing parameters of a safety-critical task in a hierarchical system. Using an adaptive technique, NosyNeighbor can improve its inference over time and evade randomization-based defense. Experimental results show that NosyNeighbor can infer victim task execution with a precision of roughly 73% under normal system load, and with a recall of about 35% using multiple malicious tasks across partitions. NosyNeighbor is also effective under the common attack model with two malicious tasks in the system, with a precision of 64%.

CCS Concepts: • **Security and privacy → Systems security**; **Embedded systems security**; **Side-channel analysis and countermeasures**.

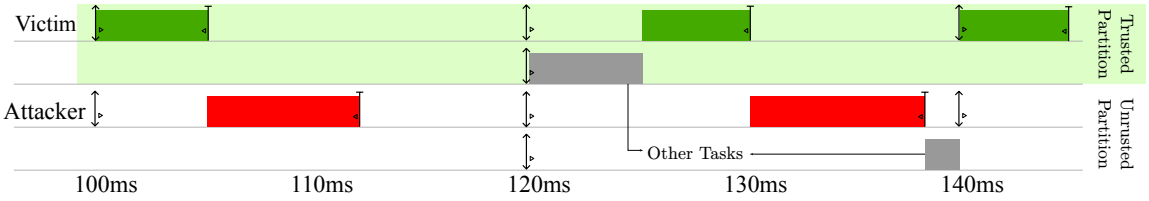
Additional Key Words and Phrases: Side-channel attacks, real-time systems, security, hierarchical systems, scheduling, randomization

ACM Reference Format:
Vijay Banerjee, Sena Hounsinou, Yanyan Zhuang, Monowar Hasan, and Gedare Bloom. 2025. On Evading Randomization-Based Defense in Hierarchical Real-Time Systems. 1, 1 (November 2025), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Real-time systems increasingly adopt hierarchical system models to enhance safety and security. These hierarchical designs are especially popular in automotive and avionic systems [3, 11, 23, 25]. In a hierarchical real-time system, tasks are grouped into partitions based on their criticality. For example, the adaptive partition scheduler in QNX [11] allocates critical system tasks and multimedia tasks into different partitions. Avionic systems extensively use hierarchical scheduling for partition-based execution, such as the compositional real-time scheduler for ARINC 653 [12].

Authors' Contact Information: Vijay Banerjee, vijay.banerjee@wsu.edu, Washington State University, Pullman, WA, USA; Sena Hounsinou, sena.houeto@metrostate.edu, Metro State University, St Paul, MN, USA; Yanyan Zhuang, yzhuang@uccs.edu, University of Colorado Colorado Springs, Colorado Springs, CO, USA; Monowar Hasan, monowar.hasan@wsu.edu, Washington State University, Pullman, WA, USA; Gedare Bloom, gbloom@uccs.edu, University of Colorado Colorado Springs, Colorado Springs, CO, USA.



(a) Using adaptive parameters in Litmus-RT with P-RES scheduling. The attacker task adapts its execution time in subsequent iterations to increase the *explored* region in the execution timeline. NosyNeighbor orchestrates such adaptive tasks to draw precise timing inferences of the victim tasks.

```
while (1) {
  x = 5;
  for(i=0; i<x; i++);
  sleep(period);
}
```
(a) A task loop

```
while (1) {
  read(pipe_fd, (void*)x, 2);
  for(i=0; i<atoi(x); i++);
  sleep(period);
}
```
(b) A task loop with adaptation

(b) An example of on-the-fly parameter adaptation. NosyNeighbor uses tasks like (b) to adjust the execution phases of malicious tasks.

Fig. 1. A demonstration of an adaptive task on Litmus-RT.

This paper presents NosyNeighbor, a novel attack against hierarchical systems that can find execution windows of a victim task in an isolated partition. In a real-time system, tasks are schedulable entities with predictable execution times and guaranteed deadlines. NosyNeighbor exploits this timing constraint of real-time systems to infer the execution of tasks across the boundaries of isolated partitions in hierarchical systems. An attacker can use NosyNeighbor to effectively evade the state-of-the-art defense techniques for hierarchical systems that depend on schedule randomization [29, 30]. These defense techniques assume tasks have static parameters (execution time and period), i.e., they presume malicious tasks exhibit the same execution behavior over time. In practice, however, the execution behavior of tasks in real systems can vary widely over time. NosyNeighbor uses malicious *adaptive* tasks to draw timing inferences of the victims that improve iteratively throughout the attack execution. Breaking the assumption that task timing is static results in a stronger yet more realistic adversarial model.

Hierarchical systems, including virtualization-based systems, use shared hardware resources for multiple tasks. The sharing of hardware makes them vulnerable to side-channel information flow between the tasks. Embedded systems are especially prone to such vulnerabilities due to limited hardware resources. In these systems, obtaining precise timing information of security-critical tasks can lead to more serious attacks such as cache side-channel attacks [26]. By using the adaptive execution model, NosyNeighbor can draw more precise timing information of a victim task than previously known, which increases the effectiveness of existing time-sensitive attacks.

Schedule-based attacks have been extensively studied in real-time systems [5, 7, 15, 17]. These attacks can be classified as anterior, posterior, concurrent, or pincer, depending on the temporal proximity of the attacker's execution in relation to the victim [21]. To determine the best timing to execute the attacker's tasks, the attacker typically starts by observing the system. During this observation, the attacker collects measurements about the system, enabling them to infer the system's parameters in general and the timing parameters of the victim task in particular. However, the existing literature does not consider *adaptive execution* of the attacker to improve the attack inference over time. NosyNeighbor uses adaptive execution to evade the state-of-the-art defense techniques that are based on schedule randomization.

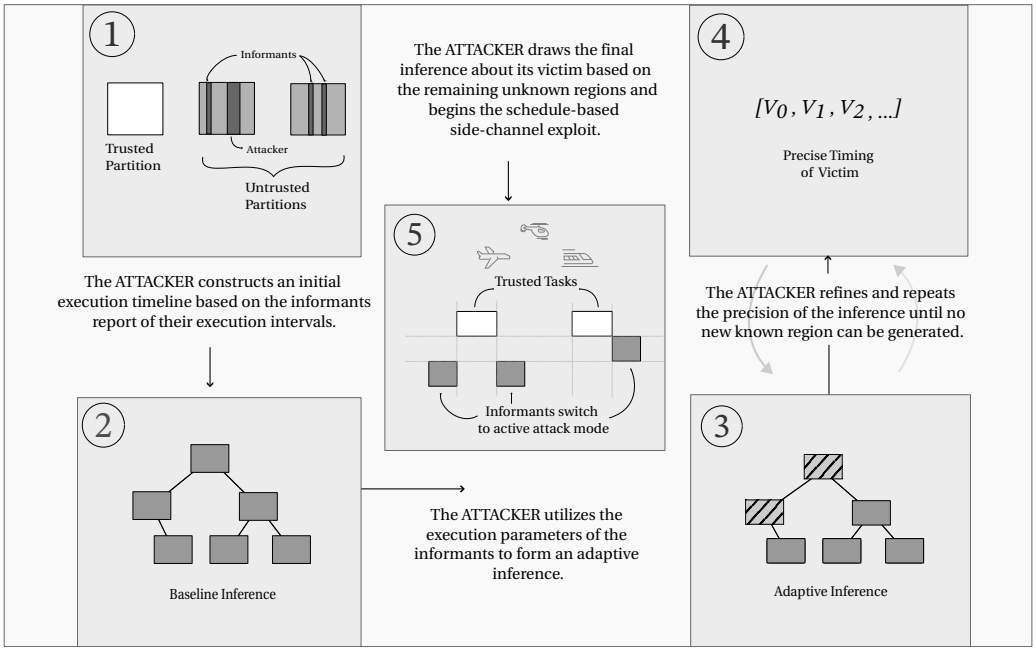


Fig. 2. The NosyNeighbor Attack: On each phase of the attack, the malicious tasks *adapt* their parameters and improve attack precision. NosyNeighbor utilizes the precise placement of malicious tasks to switch into an active attack. Section 4 discusses these steps in detail.

We illustrate the basic concepts and feasibility of adaptive attack execution in Figure 1. The execution time of the attacker task in Figure 1a is different in two subsequent executions. In the second iteration of the task that starts at 130 ms, the execution time is increased by 1 ms. This extended 1 ms execution time reflects that no higher-priority tasks were executed during that time. Such extended execution is an example of adapted execution used by NosyNeighbor to infer execution information of higher priority tasks through a side-channel. We describe such *extended execution* in Section 4.4.1. Figure 1b shows the code modification used by the attacker task of Figure 1a. Such modification can be maliciously inserted into the program by an adversary to enable the adaptive execution of a task based on external inputs. An adversary can insert such malicious programs into the system by using existing code injection or supply chain attacks [14, 22]. NosyNeighbor uses adaptive tasks similar to Figure 1b, in which we show that these malicious tasks are not required to be inside a trusted partition, and an adversary can use these tasks from an untrusted and unauthorized partition to derive the precise execution times of the victim. The attack precision depends on the relation between the actual execution window of a task and the execution window inferred by NosyNeighbor. We use attack effective window (AEW) [9, 10] to evaluate the potential success of NosyNeighbor in using the side-channel attack to execute other exploits in the system.

An overview of the NosyNeighbor attack is shown in Figure 2. First, the NosyNeighbor attacker constructs an initial timeline of the malicious tasks ① and builds a baseline inference ② of the victim task. Next, it adapts the execution parameters of the malicious tasks ③ to increase its knowledge of the system's schedule and improve the inference of the victim's execution. Over time, its inference tightens more precisely ④ on the victim's actual execution timing. Eventually, the NosyNeighbor terminates its inference ⑤ when it decides it has enough information about

the victim task to conduct a schedule-based attack. These steps use an interval tree data structure to store and parse through the victim timeline efficiently. Section 4.3.1 describes the data representation and processing by NosyNeighbor in detail.

Our Contributions. The main contributions we make in this paper are:

- We introduce NosyNeighbor, an adaptive execution model that can effectively evade schedule randomization-based defense techniques and infer precise timing information of the victim task through a timing side-channel.
- We implement a proof-of-concept to demonstrate the potential impact of NosyNeighbor and discuss the challenges and techniques for effectively implementing such an attack.
- We use the AEW concept to evaluate the attack's efficacy against a target victim task.
- We empirically evaluate the effectiveness of NosyNeighbor under different system loads and show that the adaptiveness of NosyNeighbor enables it to *pinpoint victim execution* with a precision of over 70% under normal system load and multiple malicious tasks. NosyNeighbor can achieve an attack precision of 64% with just two malicious tasks, which demonstrates its effectiveness *without* requiring a large number of malicious tasks in the system.

2 Related Work

We start with a summary of prior work on timing inference, as well as mitigation techniques against schedule-based attacks.

Parameter Inference. In ScheduLeak [5, 7], the attacker leverages an unmanned aerial vehicle's idle task to collect observations about the system's schedule from which the busy intervals and the victim task's initial offset and future arrival times are inferred. Hounsinou et al. [15] recorded controller area network (CAN) traffic to reconstruct the CAN schedule using the CAN response time analysis. They identified patterns of CAN messages that are expected to precede the victim to queue the attack message with heuristics to improve the success of their attack. Others have shown how circular auto-correlation, fast Fourier Transform [19], and periodogram [24] can be used to infer task parameters from execution traces. These approaches vary in accuracy, especially in the presence of system uncertainties such as release jitter, and have not been applied to hierarchical systems.

Schedule Randomization-Based Defense Techniques. Schedule randomization [6, 16, 28, 31] aims to reduce the apparent determinism of real-time systems to prevent an adversary from deducing timing parameters. However, Nasri et al. [21] argued that these approaches may not provide enough protection while also stating that isolation techniques (including virtualization) might directly solve the security-relevant problems of schedule information leakage. A similar line of work, Blinder [30] and TimeDice [29], use randomization-based solutions to mitigate the orchestration of multiple tasks. However, we demonstrate that a proof-of-concept NosyNeighbor is successful against these techniques. The design of NosyNeighbor makes it resilient to simple schedule randomization by considering the impact of randomization in the establishment of the baseline inference. It is an open problem whether these mitigation techniques are effective in hierarchical systems.

3 Models and Assumptions

We now present the hierarchical system model used in this work (Section 3.1) and discuss the capabilities and motivations of the NosyNeighbor attacker (Section 3.2).

Table 1. Table of notations.

Notation	Interpretation
Π	A partition
Π^k	A partition with priority and partition number k
T^k	Time period of partition k
C^k	Execution capacity of partition k
τ_i^k	Task i in partition k
t_i^k	Time period of task τ_i^k
c_i^k	Worst case execution time of τ_i^k
$\tau_{i,j}^k$	Job j of task τ_i^k
$I_{i,j,n}^k$	Informant task executed during n-th major frame of job $\tau_{i,j}^k$
ζ_n	All known regions at the end of major frame n
ζ_n'	All unknown regions at the end of major frames n

3.1 System Model

The system model used in the paper is identical to the models used in state-of-the-art systems equipped with schedule randomization-based defense [30]. In Section 5.4, we demonstrate the execution of NosyNeighbor on the same scheduler model implemented in [30], which shows the feasibility of executing NosyNeighbor in existing commercial-off-the-shelf components that are using state-of-the-art defense mechanisms in their implementations.

We assume a uniprocessor real-time hierarchical system with K partitions, among which at least one is trusted. The attacker utilizes inter-partition collusion of tasks from distinct untrusted partitions. We assume the existence of at least two untrusted partitions. This assumption is aligned with the state-of-the-art defense papers [30]. Although NosyNeighbor can be executed from a single untrusted partition if the system has only two partitions, in a multi-partition system, the attack is most effective when the malicious tasks are spread across multiple partitions. We also assume that the hypervisor is trustworthy and secure from attacks, and thus it correctly schedules partitions. Thus, a *global scheduler* is a part of the hypervisor, responsible for scheduling the partitions based on their budget. Note that the global scheduler does not have any information related to the individual tasks inside the partitions. Each partition has a *local scheduler* to handle the tasks inside that partition. As demonstrated in Figure 1a, the *P-RES* scheduler in LITMUS-RT [4] is an example of a hierarchical scheduler that matches the system model here.

Task Model. We denote a partition with priority k as Π^k, where $k \in \{0, \ldots, K-1\}$. Π^k is characterized by a period of T^k and an execution capacity of C^k within each execution period. We use the term *major frame*, denoted M, as commonly defined in hierarchical systems as the least common multiple of all T^k. Each partition Π^k contains one or more application real-time tasks. A task of priority i in partition Π^k is denoted by τ_i^k. Each task τ_i^k has a minimum inter-arrival time of t_i^k, after which a new iteration or job of τ_i^k is released. A job $\tau_{i,j}^k$ is the j'th iteration of task τ_i^k. Each task has a worst-case execution time (WCET) of c_i^k. Note that the priority of each task τ^k is unique within each partition Π^k. Table 1 provides a summary of the notations used above, along with brief descriptions for each.

3.2 Threat Model

The NosyNeighbor adversary aims to launch a schedule-based side-channel attack on a *victim task* executing in a trusted partition. We assume that the trusted partition is scheduled at a higher priority than the untrusted partitions as is typical in hierarchical systems comprising software components from multiple vendors [27]. Although the security-criticality and priority are not generally coupled in general-purpose systems, the time-critical tasks such as navigation

are generally placed at a higher priority partition in automotive-grade OS such as QNX [11]. Knowing the execution times of these time-critical tasks can enable an adversary to execute time-sensitive attacks like denial of service.

We assume that the adversary is capable of infiltrating and controlling *some* insecure application tasks executing in untrusted partitions of the hierarchical system remotely. Specifically, we assume that once the attacker has a foothold on a task, they are capable of controlling the task's execution parameters and configuring inter-partition communication for that task. This system assumption is common in real-time systems, and one can leverage existing attacks to achieve this capability [8]. We refer to the set of such tasks under the attacker's control as *malicious tasks* and denote them by $\mathcal{T}_A$. One of these tasks is designated as the NosyNeighbor, and the remainder tasks are designated *informants*—they are formally defined in Definitions 4.1 and 4.2 in Section 4.

We assume that the malicious tasks may have different requirements (in terms of real-time performance, safety-criticality, certifications, security) and therefore could be allocated to different partitions of the hierarchical system.

4 The NosyNeighbor Attack

In this section, we detail the steps involved in the NosyNeighbor attack. The NosyNeighbor adversary consists of a set of tasks, with one *observer* task, called the NosyNeighbor task, and other *informant* tasks, which are adaptive tasks similar to the one shown in the example code in Figure 1b. The NosyNeighbor attack strategy is to control these observer and informant tasks executing in untrusted partitions and collect their execution intervals to infer the timing information of the (higher priority) victim task in a trusted partition. To obtain a more precise inference, the attacker adaptively modifies the execution time and release times of the informant tasks to identify time intervals within each major frame. We formally define the properties of these tasks here.

4.1 Definitions

Definition 4.1 (The NosyNeighbor task). In the NosyNeighbor attack, a primary malicious task is responsible for recording, analyzing, and adapting the execution parameters of all the malicious tasks in the system. We refer to this task as the NosyNeighbor, which holds the following properties to execute a successful attack. The NosyNeighbor:

- Is an untrusted task.
- Has control over malicious tasks in multiple partitions, and it can communicate with the malicious tasks via inter-partition communication channels.
- Is responsible for computing the timing information of the victim.

Definition 4.2 (The Informants). The *informants* are a set of malicious tasks that are coordinated by the NosyNeighbor task. Each informant holds the following properties:

- Can access the (global) clock that synchronizes time in the system.
- Reports its own entry and exit times to the NosyNeighbor task.
- Keeps the processor busy for the execution time requested by the NosyNeighbor task.
- Has a *maximum budget*, which is the WCET allowed for its execution.

We use the notation $I^k_{i,j,n}$ to denote an informant task executed during the n'th major frame in partition Π_k, with task id i, and job id j. For instance, an informant task that executes in the temporal proximity of $\tau^k_{i,j}$ will be denoted as $I^k_{i,j}$. If the current instance of the task is executing on major frame 1, the informant will be denoted by $I^k_{i,j,1}$.

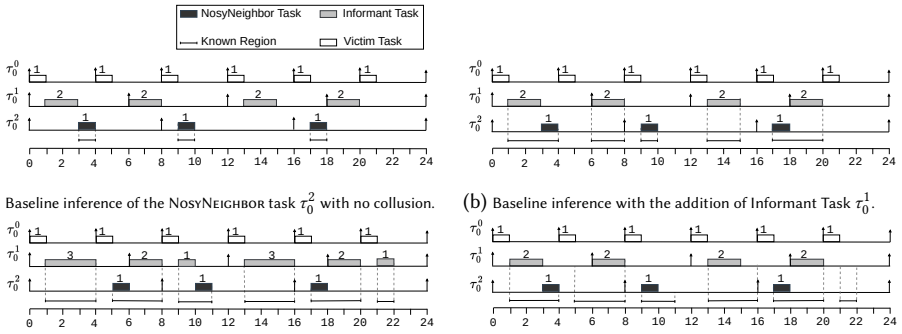
Definition 4.3 (Known and Unknown Regions). A *known region* is any time interval in a major frame during which at least one informant has executed. In contrast, an *unknown region* is an interval in the major frame during which no informant has executed since the beginning of the attack (either because no informant has been scheduled during that interval, or an execution by an informant was unsuccessful due to preemption). We denote the total known region at the end of the n'th major frame as ζ_n, and unknown regions are denoted ζ'_n. ζ'_n is considered the attack inference after n major frames.

4.2 Overview

The NosyNeighbor task uses the informants to draw precise timing information in a hierarchical system as follows:

(1) The NosyNeighbor task generates an initial set of parameters for each informant task.
(2) The informants execute based on the initial parameters and report the execution intervals of all their jobs throughout the major frame. The NosyNeighbor task constructs a baseline inference with these reports.
(3) The NosyNeighbor task refines the precision of the inference by adapting the execution parameters of some of the informants in the next major frame. The attack stops when the inference timeline stays the same for two consecutive major frames.

We refer to Steps 1 and 2 as *baseline inference* (Section 4.3) and to Step 3 as *parameter adaptation* (Section 4.4). Upon completion of the previous steps, NosyNeighbor can use the timing inference to conduct schedule-based attacks (i.e., schedule exploitation). We illustrate these stages using the following example.



(a) Baseline inference of the NosyNeighbor task τ_0^2 with no collusion.

(b) Baseline inference with the addition of Informant Task τ_0^1.

(c) Updated inference with extended execution of Informant Task τ_0^1

(d) After detecting preemption, NosyNeighbor uses shortened execution to adjust the execution of τ_0^1. The current inference now precisely predicts the start time of τ_0^0.

Fig. 3. The NosyNeighbor Adaptive Execution: the attacker adapts the known regions by modifying the parameters of the informant task τ_0^1, which improves the inference of the execution pattern of the victim task τ_0^0.

Example 4.4. Consider a three-partition hierarchical system with a major frame of $M = 24$ units as depicted in Figure 3. Task τ_0^2 is the NosyNeighbor task and τ_0^0 is the victim. The baseline inference stage begins in Figure 3a. Here, τ_0^2's knowledge about the schedule is limited to the duration of its own execution ($[3, 4)$, $[9, 10)$, and $[17, 18)$). This leads to a poor inference of the execution pattern of victim τ_0^0. When τ_0^1 is used as an informant with an initial execution budget of 2 units (Figure 3b), the known regions include additional intervals $[1, 3)$, $[6, 8)$, $[13, 15)$, and $[18, 20)$ due to the successful (non-preempted) executions of τ_0^1.

The parameter adaptation starts in the next major frame (Figure 3c). The NosyNeighbor task extends the execution time of τ_0^1 to 3 units and subsequently reduces it to 2 units (Figure 3d) because the extension resulted in preemption of $\tau_{0,1}^1$ and $\tau_{0,3}^1$ (in Figure 3c). Altogether, this yields another expansion of the known regions by generating intervals $[5, 6)$, $[10, 11)$, $[15, 16)$, and $[21, 22)$ in Figure 3c. With this, the NosyNeighbor task can compute the total known regions of M and deduce the unknown regions $[0, 1)$, $[4, 5)$, $[6, 9)$, $[11, 13)$, $[16, 17)$, $[20, 21)$, and $[22, 24)$. Six out of those seven unknown regions inferred do contain an execution of the victim task τ_0^0.

4.3 Baseline Inference

The adversary's goal in this attack stage is to establish an initial timeline. The NosyNeighbor task uses this timeline in the next attack stage for the parameter adaptation to expand the known regions. However, establishing an accurate initial timeline poses a few challenges. First, a task's execution pattern may vary from one major frame to the next due to system uncertainties, which are not considered in most state-of-the-art analyses. This can impact the attack outcome in real systems. Second, we need an efficient data structure to manage the recorded timestamps of the informant tasks and calculate the parameter adaptations for the next iterations of the attack.

4.3.1 Informant Data Representation. The NosyNeighbor attack requires storing and orchestrating the informant tasks' individual jobs over multiple major frames and analyzing their timing logs. This poses a challenge because of the number of jobs of each task that may participate in the attack. NosyNeighbor uses an interval tree [13, 20] to handle the large number of execution logs collected by informant jobs. An interval tree is an ordered self-balancing tree where each node represents an interval. It is also an efficient structure for finding overlapping intervals, which is an important property for selecting the group of informants for adaptive execution. For m informants, the tree can be constructed and searched with a time complexity of $O(m \log m)$ and $O(\log m)$, respectively.

Definition 4.5 (Informant Node and Informant Tree). An *informant node* represents a time interval that denotes the start and end times corresponding to a job's response time. We denote the informant node of an informant's job $\tau_{i,j}^k$ during nth major frame by $I_{i,j,n}^k$. The *informant tree* I is the interval tree constructed from all informant nodes. The informant tree is updated at the end of the nth major frame with new values of $I_{i,j,n}^k$.

4.3.2 Handling Uncertainty and Schedule Randomization. Schedule-based randomization adds uncertainty to the inferred timeline from the attack. Such schedule uncertainties can also result from system noises and release jitters. Depending on the randomization protocol used in the system, each job can have varying magnitudes of uncertainties.

To account for these uncertainties, the known region reported by an informant during its interval, which spans across multiple major frames, is collected to form a wider window of the known region that accommodates the variances in the timing of the tasks. Thus, the NosyNeighbor attack utilizes the sequences of major frames to discover new known regions uncovered through such observations to establish the baseline inference.

In two subsequent major frames, the reported known regions overlap only if the shift in the execution is less than the execution cost of the informant node. Hence, for major frames n and $n-1$, the reported known regions overlap if the difference between release times of $\tau_{i,j,n}^k$ and $\tau_{i,j,n-1}^k$ is less than c_i^k. The newly reported region may yield a new known region, which is wider than the previously recorded known region. The union of the overlapping intervals is recorded in an interval tree to expand the known regions from one major frame to the next. By extending the observed informant execution time over multiple major frames, we can identify the precise windows during which the informants are not allowed to execute, indicating that a victim job might have executed during that interval.

The informant node of overlapping intervals at the end of the nth major frame ($n \geq 1$) is defined as:

$$I^k_{i,j,n} = [\min\{I^k_{i,j,n}.start, I^k_{i,j,n-1}.start\}, \max\{I^k_{i,j,n}.end, I^k_{i,j,n-1}.end\}] \quad (1)$$

where $I^k_{i,j,n}$ denotes the interval recorded for job $\tau^k_{i,j}$ at the end of the nth major frame, with $I^k_{i,j,1}$ being its first recorded interval node.

Similarly, the NosyNeighbor can evade schedule randomization-based defense techniques by relying on multiple major frames to collect a baseline value for each $I^k_{i,j,n}$. The randomness achieved by most randomization-based mitigation techniques is limited because of the time-bound constraints of the real-time tasks. As a result, it is possible to reveal the predictable randomization patterns from the observation of multiple major frames.

The resulting known regions in the interval tree can potentially spread across different major frames. Thus, to ensure that the initial timeline generated takes into consideration the possibility of randomization in the hierarchical system, we proceed by generalizing Equation (1) as follows. In the n'th iteration, the new interval collected for $\tau^k_{i,j}$ will fall under one of these three cases:

(1) $I^k_{i,j,n} \succ I^k_{i,j,n-1}$
(2) $I^k_{i,j,n} \prec I^k_{i,j,n-1}$
(3) $\{I^k_{i,j,n}\} \cap \{I^k_{i,j,n-1}\} \neq \emptyset$

The succeeds notation (" $\succ$ ") denotes that the entire range on the left side is greater than the entire range on the right side of the symbol, e.g., $[3, 5] \succ [1, 2]$. Hence, Case 1) means that in the nth iteration, the informant job $\tau^k_{i,j}$ observed a later start and end time, i.e., the interval shifted later in the cycle. Case 2) means the informant's interval shifted earlier in the cycle, and a shorter response time was observed, assuming the same release time for the informant. In Case 3), the new interval overlaps with the previous interval. The set notation in case 3) shows that the intersection of the two intervals is not null.

4.3.3 Constructing the Baseline Inference. We take a heuristic approach to update the known regions after each major frame using Equation (1) by introducing additional informant nodes for cases 1) and 2) above for the same job. Since NosyNeighbor aims to segment the timeline into known and unknown regions, adding more informant nodes will cover more regions of possible executions of informants. Algorithm 1 shows the steps to construct and update the informant tree over multiple major frames.

To construct the interval tree, Algorithm 1 compares the current interval ($I^k_{i,j,n}$) to the total execution time of the informant task, which indicates preemption (Line 4). In Line 4, the condition checks if the difference between the end time δ_{end} and start time δ_{start} is greater than the execution time δ_{etime}. If the actual execution time is longer than the expected execution time, the task was likely preempted by a higher-priority task.

Subsequently, the algorithm checks whether the previously recorded interval in the set of known regions ζ overlaps with the current interval ($I^k_{i,j,n}$) (Line 7). We check overlapping intervals to handle the variances and randomizations by segmenting the informant intervals to augment the known and unknown regions or expand the overlapping regions. On Line 9, ζ is updated as the union of both intervals and inserted in the tree. Otherwise, $I^k_{i,j,n}$ is recorded in ζ' tree. The attacker then investigates whether the execution of a higher-priority secure task or a higher-priority informant task causes the overlapping preempted intervals. Line 15 first checks whether there is any preemption. If a preemption was noted, Line 16 adds the intersection window into the set of known regions, since in this case the preemption on

Algorithm 1 Forming the known and unknown set of regions

1: **Input:** $I^k_{i,j,n}$, ζ_{n-1}, ζ'_{n-1}
2: **Output:** I, ζ_n, ζ'_n
3: $\delta \leftarrow I^k_{i,j,n}$
4: **if** $\delta_{end} - \delta_{start} > \delta_{etime}$ **then**
5: $\quad \delta_{preempted} = True$
6: $O \leftarrow \text{Overlap}(\zeta'_{n-1}, \delta)$[1]
7: **if** $O = \emptyset$ **then**
8: $\quad$ **if** $\delta_{preempted}$ **then**
9: $\quad\quad \zeta'_n \leftarrow \zeta'_{n-1} \cup \{\delta\}$
10: $\quad$ **else**
11: $\quad\quad \zeta_n \leftarrow \zeta_{n-1} \cup \{\delta\}$
12: **else**
13: $\quad$ **for** $\delta' \in O$ **do**:
14: $\quad\quad$ **if** $\delta_{preempted}$ **then**
15: $\quad\quad\quad$ **if** $|\{\delta'\} \cap \{\delta\}| \neq \emptyset$ **then**
16: $\quad\quad\quad\quad \zeta_n \leftarrow \zeta_{n-1} \cup \{\{\delta'\} \cap \{\delta\}\}$
17: $\quad\quad\quad$ **else if** $|\{\delta'\} \cap \{\delta\}| = \emptyset$ **then**
18: $\quad\quad\quad\quad \delta'' = [\min\{\delta_{start}, \delta'_{start}\}, \max\{\delta_{end}, \delta'_{end}\}]$
19: $\quad\quad\quad\quad \zeta_n \leftarrow \zeta_{n-1} \cup \{\delta''\}$
20: $\quad\quad$ **else if** $\delta \succ \delta'$ or $\delta \prec \delta'$ **then**
21: $\quad\quad\quad \zeta_n \leftarrow \zeta_{n-1} \cup \{\delta\}$

one informant was likely caused by the other intersecting informants. Line 17 shows the condition where a preemption has happened, but the preemption does not involve another informant. In this scenario, the entire interval of both the involved informants is included in the inference tree, as there is a possibility of a victim's execution during this interval, but the precise timing is unknown. Line 20 accounts for the cases where there is an overlap, but no preemption.

The next step is to identify gaps in the informant tree I containing the baseline inference of the windows of the victim task. The informant nodes in I are the known regions. Since the *known* and *unknown* regions segment the timeline, all the regions in the timeline that are not covered by the informant nodes are the unknown regions. Hence, to form a precise inference, NosyNeighbor needs to identify smaller intervals within unknown regions in ζ' where the victim task might be executing.

Inference Tree Construction. The inference is also represented as an interval tree (termed *inference tree* herein). To construct the inference tree, NosyNeighbor must identify the different patterns of *gaps* in the execution intervals of the informant nodes. Some of these execution gaps emerge from preemption by higher priority partitions, or higher priority tasks in the same partition. Using the data fields from each informant node, NosyNeighbor categorizes each job as *preempted* or *non-preempted*. An informant job is considered preempted if the elapsed (wall) time is longer than its execution time.

Using Algorithm 2, NosyNeighbor identifies different patterns of victim nodes based on the informants' preemption categories. For ***non-preempted informants***, no higher-priority task has executed within their execution interval. The corresponding victim nodes are located in the gaps between such informant nodes. Formally, for an informant node $I(i) \in I$, where i denotes an arbitrary informant node in the in-order traversal of the informant tree I, the corresponding

[1] `Overlap(I, δ)` returns the nodes in interval tree I that intersect with the interval δ.

Algorithm 2 Deriving Inference from Unknown Regions

1: **Input:** I, ζ, ζ'
2: **Output:** V
3: $V = \emptyset$
4: **for** $\delta \in I$ **do**
5: $executed \leftarrow 0$
6: $O \leftarrow \text{Overlap}(I, \delta)$
7: $\delta' \leftarrow \text{NextNode}(I, \delta)$[2]
8: $V \leftarrow V \cup \{[\delta_{end}, \delta'_{start})\}$
9: **if** $O \neq \emptyset$ & $\delta_{end} - \delta_{start} > \sum o \in O + \delta_{etime}$ **then**
10: $budget \leftarrow \delta_{etime}$
11: **if** $|O| = 1$ **then**
12: $executed = \delta_{end} - \delta'_{end}$
13: $remaining \leftarrow budget - executed$
14: **else**
15: **for** $i \in [1, O_{end})$ **do**
16: $executed \leftarrow executed + (O_{i+1}.start - O_i.end)$
17: $executed \leftarrow executed + \delta_{end} - O_{last}.end$
18: $remaining \leftarrow budget - executed$
19: **if** $O_1.start - \delta_{start} \neq remaining$ **then**
20: $V \leftarrow V \cup \{[\delta_{start} + remaining, O_1.start\}$
21: **return** V

inference nodes δ_v and δ'_v are:

$$\delta_v = [I(i-1)_{end}, I(i)_{start}]$$
$$\delta'_v = [I(i)_{end}, I(i+1)_{start}] \tag{2}$$

Equation (2) can yield interval endpoint values that are 0 for scenarios when the next executing task in the timeline is one of the informant tasks because the start and end times are consecutive. In these scenarios, δ_v is not added to the inference tree V. Line 8 in Algorithm 2 corresponds to Equation (2) above. The intuition is to assume that the interval where no informant is executing is also a potential victim inference. This case is visualized in Figure 6.

For ***preempted informants***, there can be two cases: (1) multiple informant tasks overlap (Line 10), (2) no other informant task overlaps (Line 14). In case (1), the preemption is caused by one of the other informants, a higher-priority task, or both. To find the victim node inference in this case, we use the following equation for an arbitrary informant node $I(i) \in I$:

$$\Delta = I(i) \setminus \{I(j) \in I \mid I(i) \cap I(j) \neq \emptyset\} \tag{3}$$

Equation (3) takes the interval of $I(i)$ and excludes (via $\setminus$) the intervals covered by other informant nodes. Thus, Δ is a set with intervals as elements. Each of these intervals is added to V' as an inference node. Line 12 implements Equation (3) to select the time regions where the informant task got preempted, leaving *gaps* in the execution phase that can be potential execution times of the victim. These gaps are then added to the inference tree V. In Equation 3, the time interval of an inference node is treated like a set with each element corresponding to a discrete sub-interval within the inference node. The Δ is the set of such sub-intervals where none of the overlapping informants were executing.

[2] `NextNode(I, δ)` returns the interval node after δ in the in-order traversal of interval tree `I`.

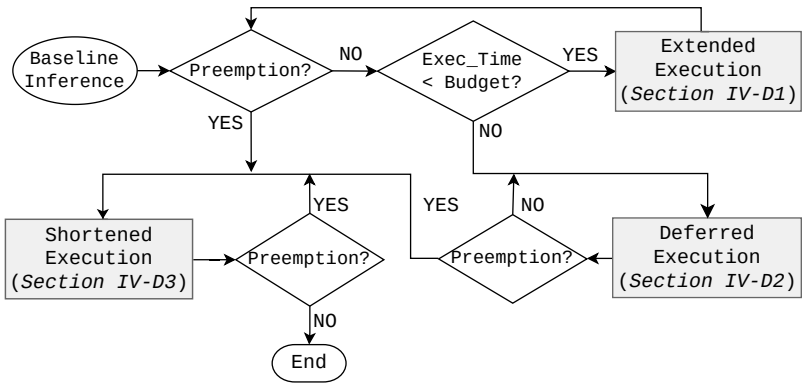


Fig. 4. Informant Task Execution Model.

These sub-intervals are the gaps in the execution. This case is visualized in Figure 10b, where τ_0^1 and τ_1^1 are overlapping informants, but neither of these overlapping tasks were executing during the sub-interval $[80ms, 90ms)$.

4.4 Parameter Adaptation

After forming the baseline inference, NosyNeighbor uses multiple adaptive strategies to adapt the execution parameters of informant tasks to improve the total timeline area covered by all the intervals in V. The goal of these strategies is to *probe* the victim nodes by executing informant tasks during time intervals that should overlap with the victim nodes by extending, deferring, or shortening the execution of an informant, as shown in Figure 4. To effectively probe the victim nodes, NosyNeighbor first selects a subset of informant nodes and sends the modified task parameters to them for adaptive execution in the next major frame.

Probes have two possible outcomes: the informants can either execute in overlapping intervals with each other, or the informants get preempted. NosyNeighbor updates the victim tree V using Equations (2) and (3) in these two scenarios. Algorithm 2 infers the time intervals where multiple informant nodes overlap. The victim tree derived from Algorithm 2 is improved by shortening the time interval of each node in V during subsequent major frames. After each major frame, the informant tree is updated using Algorithm 1.

During the construction of the baseline inference, NosyNeighbor uses combinations of informant nodes to find gaps in execution times. To overlap adaptive execution with a victim node, $\delta_v \in V$, NosyNeighbor selects the nodes from I that were used for calculating the time window δ_v. NosyNeighbor uses these selected nodes to reduce the unknown regions using different strategies based on whether the victim node was generated by non-preempted or preempted informants. Each victim node is marked as *visited* when the maximum possible informant tasks have been attempted to execute within that interval. NosyNeighbor uses three strategies to *visit* the victim nodes: extended execution, deferred execution, and shortened execution. Adaptive execution finishes after the victim node intervals is not shortened by subsequent attack iterations, i.e., after one major frame with no change in the victim node. To keep track of the interval updates in each iteration of the attack, the structure of δ_v contains the list of informant tasks, and indicate if this node has been visited, and the victim interval (start to end time). We next describe each of the three adaptive strategies for visting victim nodes in detail.

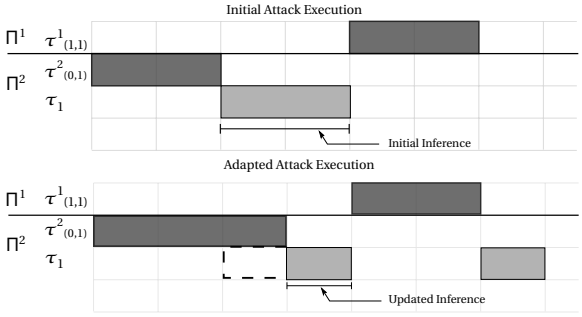


Fig. 5. Extended execution of informant $\tau^1_{0,1}$.

4.4.1 Extended Execution. Each task in the system has a maximum execution budget. In this strategy, the informant does not initially consume its entire budget. Instead, it only uses a portion of its budget for the baseline inference, which enables it to increase its execution time during adaptive execution. Extended execution is used when the victim node resides between the *end* time of a non-preempted informant and the *start* time of another informant.

Extended execution of a task from a higher priority partition will block the execution of a task from a lower priority partition. Hence, if a victim node emerges due to a task from a partition of lower priority, extended execution of a higher priority task will block it, and the victim node will be shortened or removed from the victim tree, improving precision. Example 4.6 illustrates the extended execution strategy.

Example 4.6. Consider Figure 5 for this example.We denote the victim node as τ^0_0. Suppose τ^0_0 delayed the release of $\tau^1_{0,1}$. After the baseline inference, NosyNeighbor modifies the parameter of informant $\tau^1_{0,1}$ to extend its execution time. Due to the higher priority of Π_2 over Π_1, the informant node gets shifted due to the extended execution of $\tau^1_{0,1}$. Since the informant was able to execute for the entire window without preemption, we remove its informant node from the victim tree V. The resulting inference is smaller (more precise), and therefore, the likelihood of predicting the victim task's execution is greater.

4.4.2 Deferred Execution. Each task in the system has a maximum execution budget that gets replenished at regular intervals. A task is allowed to be executed anywhere in the timeline as long as it does not exhaust the allocated budget. If the allocated budget is exhausted, the task has to wait for the next replenishment period to start execution again. We use this property of budgets to defer the execution of informants.

Deferred execution can be achieved by calling `sleep()` from the informant task to allow the informant to probe a victim node interval. This is highly effective if the informant has already exhausted its allocated budget. An example of deferred execution is shown in Figure 6. Similar to Example 4.6, the victim node will be removed from V after the adapted execution of $\tau^1_{0,1}$.

4.4.3 Shortened Execution. Section 4.3.3 describes the two cases of preempted informant tasks. In both cases, shortened execution enables NosyNeighbor to find a more precise starting point of the respective victim node. From the constructed interval tree and interval data in Example 4.4, the interval $[78, 90]$ has a length of 12. However, the task's allowed execution time is 6. Hence, the informant was preempted. According to the preliminary inference strategy (Section 4.3.3), this interval will be treated as a victim node and will be added to V. To find out the precise starting point

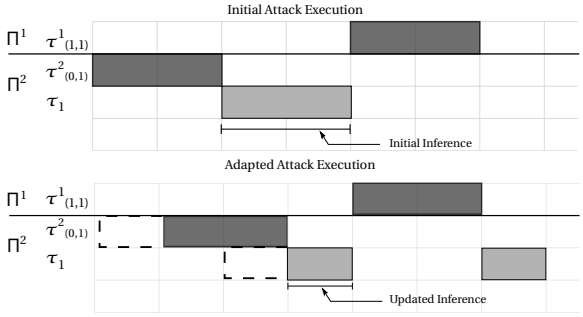


Fig. 6. Delayed Execution of informant $\tau^2_{(0,1)}$.

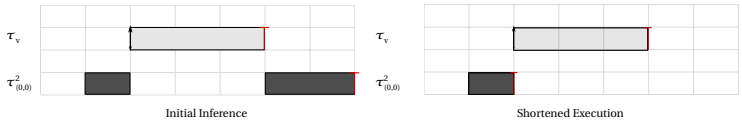


Fig. 7. Shortened execution of informant $\tau^2_{(0,0)}$.

of the preempting task, NosyNeighbor will instruct the informant to reduce its execution time by 1 unit in subsequent iterations until the requested execution time matches the actual execution time.

Figure 7 shows the schedule diagram of [78, 90]. In this example, $\tau^2_{0,0}$ was preempted by the victim task, and the shortened execution strategy placed the informant immediately before the victim task's execution. Such placement enables NosyNeighbor to escalate the passive attack into an active *anterior attack* [21]. Interestingly, iteratively shortening the execution time also reveals the entire interval of τ_v, which can be calculated as $\tau_v = [I^2_{0,0,n}.end, I^2_{0,0,n-1}.end - 1]$, if the execution time is shortened by one unit in each iteration.

Summary & Takeaway. Taken together, parameter adaptation strategies enable NosyNeighbor to adapt the informants to grow, shift, or shrink their execution times iteratively across major frames. This adaptive execution is a key differentiator that enables the NosyNeighbor attack to succeed. Furthermore, the lack of consideration for such adaptation in the existing countermeasures to prevent timing inference makes them ineffective at preventing a NosyNeighbor attack.

5 Evaluation

We evaluated NosyNeighbor on synthetic task sets to validate our approach and simulate the impact of NosyNeighbor on a range of realistic real-time systems. The evaluation shows the effectiveness of NosyNeighbor in detecting the execution of a victim task. Section 5.1 details the experimental setup for NosyNeighbor, the metrics for evaluation are described in 5.2, and Section 5.3 presents our results. Note that the following evaluation assumes that the victim task has the highest task priority in the system. The assessment results would vary based on the priority of the victim task relative to the priority of the other functions in the system, including that of the informants. Since the main focus of this paper is to introduce the attack technique and its dependency on the informant parameters, the evaluation of the attack impact based on the victim's priority and the scheduling algorithm used is outside the scope of this paper and could be a future work.

5.1 Experimental Setup

To evaluate the impact of NosyNeighbor, we used an iterated form of the UUniFast [2] algorithm. We first generated the parameters for each partition and then used UUniFast to generate the local tasks for each partition.

In the following experiments, we set M as 1000. The period of each partition is randomly selected from the set of factors of M. Hence, M is divisible by $T^k \forall k$. Each partition's utilization U^k is then chosen by the UUniFast algorithm with a target utilization of 60% unless otherwise specified. The budget of each partition is $U^k \times T^k$. Using the budget as the maximum possible execution of all tasks combined, we then used UUnifast to derive the utilization vector for the partition's tasks with periods that are randomly picked from the set of factors of T^k. Similar to the partition, we used each task's period and utilization to assign it a budget. The priorities of all the tasks within each partition are set using rate monotonic scheduling [18].

Among the set of all tasks, we randomly pick a (possibly empty) subset of tasks from each partition, except the highest priority partition, as informant tasks. The total number of informants picked from each partition is also randomly chosen for each partition.

Using these generated hierarchical tasks, we developed a *discrete event simulator*. In each iteration, we use K event queues, one for each partition, to select the highest priority task among all the partition queues and simulate execution. At each clock tick, all these executions are recorded in a *timeline*. This discrete event simulator creates a similar timeline model to that of a real system's execution. Furthermore, a randomized priority inversion is used to replicate the schedule randomization defense. State-of-the-art schedule randomization, such as Blinder, implements similar randomized priority inversion to reduce the precision of the inference from a timing-based side-channel attack.

5.2 Evaluation Metrics

We evaluate the efficacy of NosyNeighbor using precision and recall, which are widely used metrics for classifier evaluations. We use the following definitions of precision and recall for our experiments.

Precision: We define precision as a percentage of the correct guesses out of all the nodes reported as potential victim windows. Formally,

$$\text{Precision} = \frac{|\text{True Positive}|}{|\text{True Positive + False Positive}|}. \tag{4}$$

Note that a low precision indicates a high number of falsely identified victim execution windows.

A detected window is considered a *True Positive* if the detected window is within the attack effective window (AEW) of the secure task phase. We identify true positives by taking the intersection of NosyNeighbor's generated inference tree and an interval tree constructed from the actual execution intervals of the victim task. Here precision is a proxy for the likelihood that an execution interval identified as being the victim's is, in fact, the victim's. We first find the intersection of the inference tree and the original execution times of the trusted task and divide it by the total number of reported victim inferences in ζ'.

Recall: Recall shows the percentage of the correctly identified victim inference windows out of all the actual victim execution windows. This metric helps in understanding the quality of the predicted inference tree. We measure the recall using the following definition:

$$\text{Recall} = \frac{|\text{True Positive}|}{|\text{True Positive + False Negative}|} \tag{5}$$

Note that a low recall indicates missed opportunities, while a high recall indicates the attack finds most of the victim's execution.

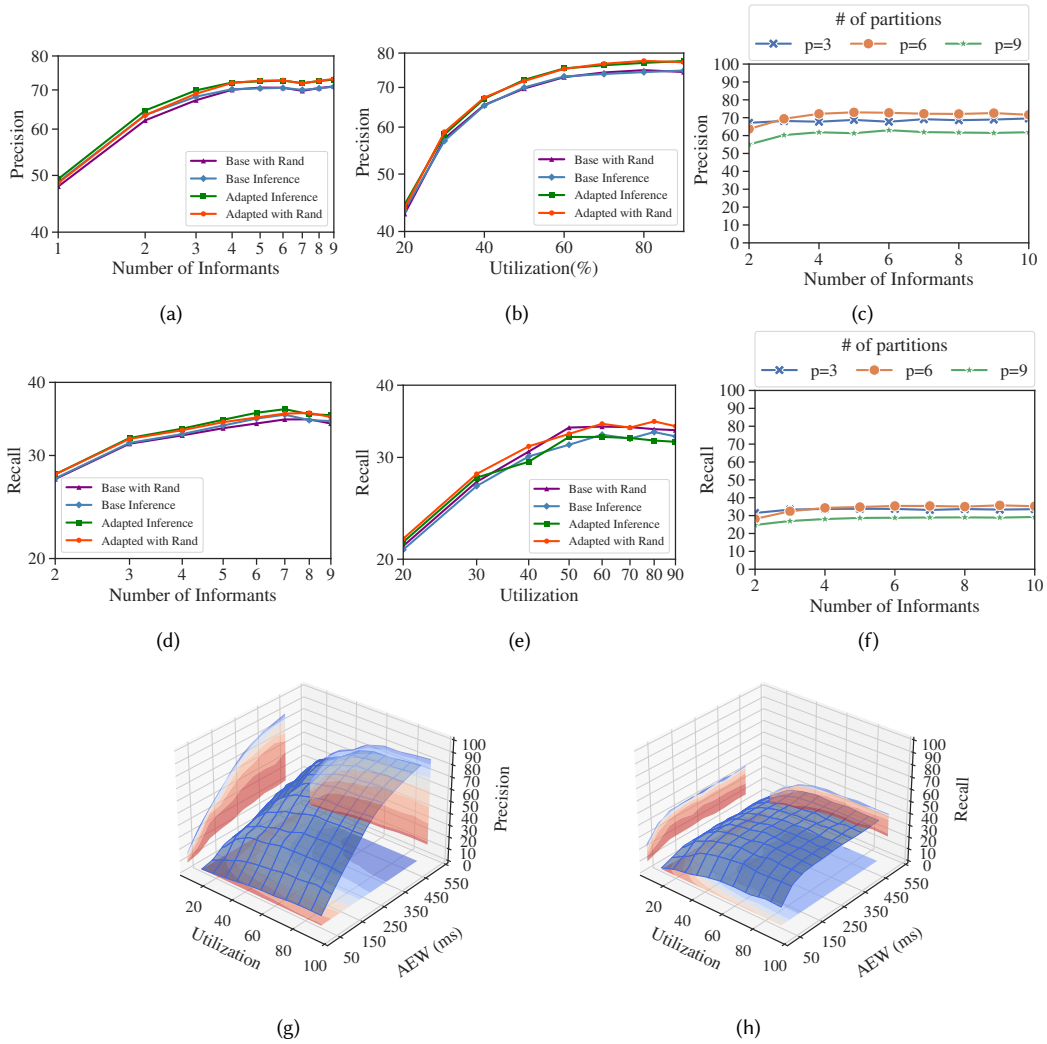


Fig. 8. NosyNeighbor Evaluation: 8a and 8d shows performance with different number of informants. 8f and 8c extends the evalulation to include partition count. Figures 8b, 8e 8g 8h show the effectiveness of NosyNeighbor under different system loads and AEW.

5.3 Experiments and Results

We have two categories of experiments. First, we perform a parameter space exploration to understand the impact of different task parameters of the system on the quality of NosyNeighbor's prediction. The performance of NosyNeighbor at different task parameters demonstrates the correctness of NosyNeighbor under normal system conditions.

In the second set of experiments, we evaluate the performance based on attack parameters. Note that the adversary does not decide NosyNeighbor's attack parameters used in the experiments; instead, these parameters are linked to an existing system's vulnerability that the attacker exploits.

Impact of Informant Count. We first evaluate the impact of the number of informants on the inference quality. We run NosyNeighbor on 10, 000 random task sets generated using the setup described in section 4. The system utilization is set to 60%, with 6 partitions, out of which 3 partitions are impacted. Each partition has 4 tasks. The informant tasks are randomly selected from the affected partitions. Figure 8a shows there is no substantial impact of the number of informants on the inference precision with the given system parameters. We observe that even in the presence of a randomization-based defense, NosyNeighbor can achieve minimal impact due to the adapted execution. The base inference is established after repeated observation and correction of the inference timeline over multiple hyperperiods, allowing the base inference to be precise on its own. The adapted inference further improves performance over the initial base inference, as shown in Figure 8b. In a real-time system, even a slight improvement in precision can significantly influence the overall outcome of the attack.

Impact of Number of Partitions. We further extended this experiment in Figure 8f and Figure 8c, which shows that increasing the total number of partitions in the system does not have a significant impact on NosyNeighbor's performance. A higher number of partitions in the system adds more randomization in the selection of affected partitions. Hence, the performance analysis observed in Figures 8f and 8c shows that the performance of NosyNeighbor does not depend on the combination of partitions where NosyNeighbor executes. An adversary can leak the victim execution intervals with a precision of over 60% irrespective of the number and placement of affected partitions or informant tasks in the system.

Impact of Utilization. System utilization determines the amount of noise that NosyNeighbor will face during the execution. For higher utilization systems, NosyNeighbor will have longer windows and higher interruption from tasks other than the victim tasks. From the previous experiments, we observed that the precision value does not improve significantly beyond 4 informants. Following this result, we set the number of informants to 4 in this experiment. We also set the number of affected partitions to 3 with a total of 6 partitions in the system. The guess precision also depends on the size of the AEW for the victim task. To reduce the impact of AEW in this experiment, we set the value of AEW to 500, which enables us to see the impact of utilization without the AEW constraints.

Figure 8b and Figure 8e show the result of this experiment. Notice that the precision and recall improve with an increasing utilization. We observed that at lower utilization, the informants exhaust their budgets or meet their deadlines during parameter adaptation. However, they are unable to pinpoint the precise window of the victim due to the system's lower utilization. The derived windows in lower utilization levels are wide and outside the AEW range. In the case of higher system utilization, where the schedule is *denser*, NosyNeighbor is able to narrow the inference window by adapting the execution parameters. With a system utilization of 60%, the average precision is over 70%. 60% utilization is realistic for many real-time systems, and this experiment shows that NosyNeighbor can effectively infer victim execution under normal system loads.

Attack Effectiveness. We combine the results from all previous experiments to analyze the effectiveness of NosyNeighbor inference. The AEW determines whether or not the inference window can be utilized to execute an active attack on the victim task. To study the attack's effectiveness, we plotted the precision and recall performance of NosyNeighbor at different levels of utilization and AEW length.

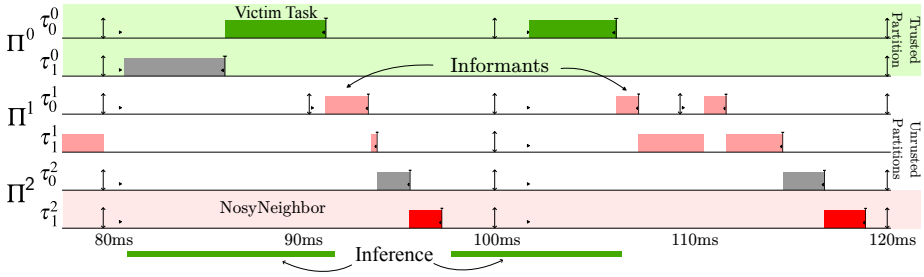


Fig. 9. Executing NosyNeighbor in Litmus-RT under P-RES scheduler. The execution trace shows the inference drawn by NosyNeighbor using the execution timestamps of the informants τ_0^1 and τ_1^1.

In Figure 8g, we observe that AEW can significantly impact the inference precision. This implies that for systems with very tight AEW, NosyNeighbor can have many false positives, which lowers the precision score of NosyNeighbor. However, it can still reach a precision of 30% at 60% utilization and AEW of 50 ms.

We observe in Figure 8h that the recall value is around 30% even for a wide AEW. This implies that NosyNeighbor is unable to identify all the execution windows of the victim task. This observation is the result of tasks with priority higher than NosyNeighbor and corresponding informants. However, even if a small subset of the attack is successful, a safety-critical system can have a catastrophic impact.

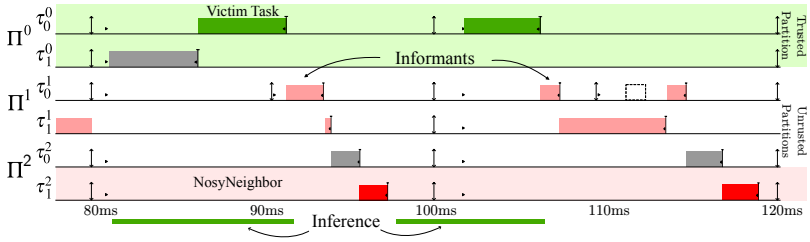
5.4 Case Study: NosyNeighbor Peeks Through the Blinds

We conducted a proof-of-concept NosyNeighbor attack on Litmus-RT. We used the P-RES [4] scheduler, which is a partition-based scheduler widely used in the literature for testing and validation of partition-based schedulers. In Blinder [30], the state-of-the-art randomization-based defense in hierarchical systems, the P-RES scheduler is extended to prevent timing side-channels by avoiding interference between tasks. The extended scheduler implementation is termed P-RES-NI. The P-RES-NI scheduler prevents task interference using a calculated priority inversion of higher-priority tasks.

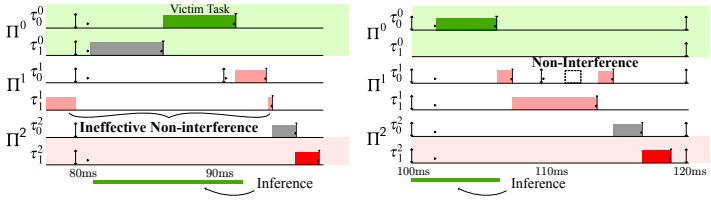
Figure 9 shows the execution trace of NosyNeighbor adaptive attack on a sample task set under the default P-RES scheduler. Tasks τ_0^1, τ_1^1, and τ_0^2 are the informant tasks in the system, sending their execution timestamps to the NosyNeighbor task τ_1^2. The baseline inference derived by NosyNeighbor in the given execution trace shows that the victim task executed during the time windows $[81ms, 91ms]$ and $[99ms, 105ms]$. From the task set, we observed that the derived baseline inference is correct. NosyNeighbor will further improve the length of the inference window in the subsequent stages by adapting the parameters of τ_0^1 and τ_1^1 using the flow chart shown in Figure 4.

To prevent this kind of coordination of malicious tasks, Blinder shifts executing a higher priority task to prevent forming interference patterns on the lower priority tasks. In Figure 10a, we implemented the P-RES-NI scheduler from Blinder and executed NosyNeighbor on the same task set as in Figure 9.

We observed two limitations of Blinder in our task model that make Blinder vulnerable to adaptive attacks. First, in Figure 10b, we observed that Blinder is ineffective due to the tight deadline. The tighter deadline of task τ_1^1 at 100ms does not allow enough window for priority inversion without forcing the other tasks to miss the deadline. As a result, in Figure 10b, the informants could still create interference patterns. NosyNeighbor uses Algorithm 2 to derive the inference window shown in Figure 10b. Note that the end of the inference window precisely coincided with the ending



(a) Executing NosyNeighbor in the presence of randomization-based defense with P-RES-NI scheduler



(b) (c)

Fig. 10. The NosyNeighbor attack execution in the presence of Blinder non-interference scheduler. 10b, and 10c are zoomed-in portions of 10a.

of the victim task τ_0^0's. Hence, an adversary can use the given inference window to successfully execute an anterior attack [21] on the victim task even under the presence of randomization-based defense.

The second limitation of Blinder is highlighted in Figure 10c. We observed that the P-RES-NI scheduler was successful in preventing the formation of interference patterns through calculated priority inversion. However, when NosyNeighbor receives the execution time stamps of τ_0^1 and τ_1^1 at 115ms, NosyNeighbor uses Algorithm 1 to derive the baseline inference by combining the timestamps of the informant tasks to generate the victim's timing inference without using the preemption patterns between tasks.

6 Conclusion & Future Directions

NosyNeighbor shows that randomization-based defenses may be ineffective in real-time systems against adaptive attack techniques where an attacker can modify its execution parameters. To our knowledge, NosyNeighbor is a first-of-a-kind novel schedule-based attack surface for hierarchical systems that shows application tasks, even in *trusted partitions are vulnerable* to schedule inference and therefore *schedule-based attacks* despite utilizing strong isolation guarantees.

To counter adaptive attacks like NosyNeighbor, the effectiveness of deterministic defense techniques should be further investigated. Other kinds of moving target defense such as restart-based techniques [1] can effectively increase the AEW and remove all recorded timing information that the attacker uses in subsequent iterations. Using restart-based techniques will force the attacker to start the attack execution from the beginning. Further research is needed to assess the effectiveness and cost of such defense techniques to defend against adaptive attacks.

The empirical evaluation shows that the effectiveness of NosyNeighbor is impacted by the AEW. Using some deterministic techniques to decrease the AEW can reduce the impact of NosyNeighbor by providing less time to execute other time-sensitive attacks during that window. Existing work that flushes the task information from the

shared resources before the execution of untrusted tasks [9, 10] might be effective in reducing the impact of timing attacks. However, most prior work only focus on non-hierarchical systems, and some of these techniques are only targeted toward confidentiality. Using NosyNeighbor to launch timed DoS or false data injection attacks can impact availability or integrity, which are often critical in real-time systems. Future research can extend the existing techniques to hierarchical systems and also investigate ways to prevent compromising system availability and integrity.

7 Acknowledgements

We thank the anonymous reviewers for their helpful suggestions. This work is supported by NSF 2138295 and NSF 2442595. Any findings, opinions, recommendations, or conclusions expressed in the paper are those of the authors and do not necessarily reflect the sponsor's views.

References

[1] Fardin Abdi Taghi Abad, Renato Mancuso, Stanley Bak, Or Dantsker, and Marco Caccamo. 2016. Reset-based recovery for real-time cyber-physical systems with temporal safety constraints. In *2016 IEEE 21st ETFA*. 1–8. https://doi.org/10.1109/ETFA.2016.7733561

[2] Enrico Bini and Giorgio C Buttazzo. 2005. Measuring the performance of schedulability tests. *Real-Time Systems* 30, 1 (2005), 129–154. Publisher: Springer.

[3] Gedare Bloom and Joel Sherrill. 2020. Harmonizing ARINC 653 and Realtime POSIX for Conformance to the FACE Technical Standard. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*. 98–105. https://doi.org/10.1109/ISORC49007.2020.00023 ISSN: 2375-5261.

[4] Bjorn B Brandenburg. 2011. *Scheduling and locking in multiprocessor real-time operating systems*. Ph. D. Dissertation. The University of North Carolina at Chapel Hill.

[5] Chien-Ying Chen, Amiremad Ghassami, Stefan Nagy, Man-Ki Yoon, Sibin Mohan, Negar Kiyavash, Rakesh B Bobba, and Rodolfo Pellizzoni. 2015. *Schedule-based side-channel attack in fixed-priority real-time systems*. Technical Report.

[6] Chien-Ying Chen, Monowar Hasan, AmirEmad Ghassami, Sibin Mohan, and Negar Kiyavash. 2018. Reorder: Securing dynamic-priority real-time systems using schedule obfuscation. *arXiv preprint arXiv:1806.01393* (2018).

[7] Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B. Bobba, and Negar Kiyavash. 2019. A Novel Side-Channel in Real-Time Schedulers. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 90–102. https://doi.org/10.1109/RTAS.2019.00016

[8] Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B Bobba, and Negar Kiyavash. 2019. A novel side-channel in real-time schedulers. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 90–102.

[9] Jiyang Chen, Tomasz Kloda, Ayoosh Bansal, Rohan Tabish, Chien-Ying Chen, Bo Liu, Sibin Mohan, Marco Caccamo, and Lui Sha. 2021. SchedGuard: Protecting against Schedule Leaks Using Linux Containers. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 14–26. https://doi.org/10.1109/RTAS52030.2021.00010

[10] Jiyang Chen, Tomasz Kloda, Rohan Tabish, Ayoosh Bansal, Chien-Ying Chen, Bo Liu, Sibin Mohan, Marco Caccamo, and Lui Sha. 2023. SchedGuard++: Protecting against Schedule Leaks Using Linux Containers on Multi-Core Processors. *ACM Trans. Cyber-Phys. Syst.* 7, 1, Article 6 (feb 2023), 25 pages. https://doi.org/10.1145/3565974

[11] Attilla Danko. 2014. Adaptive partitioning scheduler for multiprocessing system. US Patent 8,631,409.

[12] Arvind Easwaran, Insup Lee, Oleg Sokolsky, and Steve Vestal. 2009. A compositional scheduling framework for digital avionics systems. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 371–380.

[13] Herbert Edelsbrunner and Hermann A. Maurer. 1981. On the intersection of orthogonal objects. *Inform. Process. Lett.* 13, 4-5 (1981), 177–181.

[14] Badis Hammi, Sherali Zeadally, and Jamel Nebhen. 2023. Security threats, countermeasures, and challenges of digital supply chains. *Comput. Surveys* (2023).

[15] Sena Hounsinou, Mark Stidd, Uchenna Ezeobi, Habeeb Olufowobi, Mitra Nasri, and Gedare Bloom. 2021. Vulnerability of controller area network to schedule-based attacks. In *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 495–507.

[16] Kristin Krüger, Gerhard Fohler, and Marcus Volp. 2017. Improving security for time-triggered real-time systems against timing inference based attacks by schedule obfuscation. (2017).

[17] Kristin Krüger, Marcus Volp, and Gerhard Fohler. 2018. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. *LIPIcs-Leibniz International Proceedings in Informatics* 106 (2018), 22.

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

[19] Songran Liu and Wang Yi. 2020. Task parameters analysis in schedule-based timing side-channel attack. *IEEE Access* 8 (2020), 157103–157115.

[20] Edward M McCreight. 1980. *Efficient algorithms for enumerating intersecting intervals and rectangles*. Technical Report.

[21] Mitra Nasri, Thidapat Chantem, Gedare Bloom, and Ryan M Gerdes. 2019. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 103–116.

[22] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*. Springer, 23–43.

[23] Paul J Prisaznuk. 2008. ARINC 653 role in integrated modular avionics (IMA). In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. IEEE, 1–E.

[24] Şerban Vădineanu and Mitra Nasri. 2020. Robust and accurate period inference using regression-based techniques. In *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 358–370.

[25] Steven H VanderLeest. 2010. ARINC 653 hypervisor. In *29th Digital Avionics Systems Conference*. IEEE, 5–E.

[26] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. 2012. A cache timing attack on AES in virtualization environments. In *Financial Cryptography and Data Security: 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers 16*. Springer, 314–328.

[27] Richard West, Ye Li, Eric Missimer, and Matthew Danish. 2016. A virtualized separation kernel for mixed-criticality systems. *ACM Transactions on Computer Systems (TOCS)* 34, 3 (2016), 1–41.

[28] Man-Ki Yoon, Jung-Eun Kim, Richard Bradford, and Zhong Shao. 2019. Taskshuffler++: Real-time schedule randomization for reducing worst-case vulnerability to timing inference attacks. *arXiv preprint arXiv:1911.07726* (2019).

[29] Man-Ki Yoon, Jung-Eun Kim, Richard Bradford, and Zhong Shao. 2022. TimeDice: Schedulability-Preserving Priority Inversion for Mitigating Covert Timing Channels Between Real-time Partitions. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 453–465. https://doi.org/10.1109/DSN53405.2022.00052 ISSN: 2158-3927.

[30] Man-Ki Yoon, Mengqi Liu, Hao Chen, Jung-Eun Kim, and Zhong Shao. 2021. Blinder: Partition-Oblivious Hierarchical Scheduling. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2417–2434. https://www.usenix.org/conference/usenixsecurity21/presentation/yoon

[31] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. 2016. TaskShuffler: A Schedule Randomization Protocol for Obfuscation against Timing Inference Attacks in Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–12. https://doi.org/10.1109/RTAS.2016.7461362