On Timing-Based Information Leakage in Data Flow-Driven Real-Time Cyber-Physical Systems

Mohammad Fakhruddin Babar and Monowar Hasan
School of EECS, Washington State University, Pullman, USA

***Abstract*—Cyber-physical systems increasingly rely on data-driven real-time tasks whose timing behavior can inadvertently expose sensitive operational information to unprivileged adversaries. We study information leakage for a special class of such tasks that operate in two execution modes: *typical* execution (invoked most of the time) and *critical* execution (used to handle exceptional conditions). We develop a new statistical analysis technique and show that analyzing the response times of the low-priority task enables the extraction of the execution behavior of the high-priority task. We demonstrate the feasibility of inference on two cyber-physical platforms: an off-the-shelf manufacturing robot and a custom-built surveillance system.**

***Index Terms*—Information leakage, real-time systems.**

I. Introduction

Real-time systems form the foundation of safety-critical cyber-physical systems—including power plants, manufacturing, robotics, and healthcare—where timing violations can cause physical harm [1]–[4]. Many data-driven real-time tasks in such cyber-physical deployments operate in two execution modes: *(a) typical* and *(b) critical*. For example, in a healthcare monitoring system, a real-time task functions in typical mode to monitor stable vital signs with minimal processing but switches to critical mode upon detecting abnormal readings, such as a sudden drop in heart rate. In critical mode, the task runs complex algorithms and triggers alerts with longer execution times to ensure patient safety. Likewise, in an automated assembly line, the control system operates in typical mode for routine tasks such as monitoring conveyor speeds and sensors with standard processing. During anomalies, such as machinery malfunctions, it may switch to critical mode to perform complex diagnostics and emergency responses with longer execution times.

Preventing unauthorized information flow is crucial in safety-critical cyber-physical environments such as industrial control and autonomous systems. However, the predictable nature of real-time schedulers can unintentionally create opportunities for timing-based information leakage in priority-driven systems—which is especially concerning in cyber-physical systems where such leaks can enable targeted physical attacks. For example, if an attacker can determine when a critical task is about to execute, they could gather side-channel data (e.g., cache usage) or launch denial-of-service attacks to block the task [5].

We would like to thank Zain Hammadeh and Mohammad Hamad for providing us with the idea of this work. This research is partly supported by the U.S. National Science Foundation Award 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.

In this paper, we present a statistical inference model to analyze information leakage in nondeterministic dual-mode real-time systems—a threat scenario directly relevant to cyber-physical deployments where critical task behavior governs physical actuators and sensors. We show that the response time of the low-priority rogue task can be used as a leakage signal to predict the future critical arrival of the high-priority victim task. Such information leakage allows an unprivileged, low-priority task (i.e., observer task) to infer the timing behavior of a critical high-priority task (i.e., victim task) by calculating its own response time.

II. Model and Assumptions

A. System Model

We consider a uniprocessor real-time system with n periodic tasks $\Gamma = \{\tau_1, \cdots, \tau_n\}$ scheduled by a fixed-priority policy. In this work, we want to investigate whether a low-priority task (called *observer*, denoted by τ_o) can infer the behavior of a targeted, higher-priority one (called *victim*, denoted by τ_v). Let us denote $hp(\tau_i)$ as the set of tasks with higher priority than τ_i. By assumption, $hp(\tau_o) = \{\tau_1, \tau_2, \cdots, \tau_v, \cdots\}$ and τ_1 is the highest-priority task. Each task τ_i generates an infinite sequence of *jobs* and is characterized by a tuple (C_i, T_i, D_i). The first parameter, C_i, denotes the worst-case execution time (WCET). The jobs of τ_i are periodically invoked with an inter-arrival time (period) T_i and are completed before the deadline D_i. We assume the tasks follow the implicit deadline model ($D_i = T_i$), i.e., the tasks must be completed before their next arrival.

Task execution times are data-driven. Hence, depending on input or processing data, each task τ_i has two execution modes: *(a)* typical execution time, C_i^{typ} (which is the execution time during normal conditions), and *(b)* critical execution time, C_i^{cri}, that the runtime of jobs that occur depending on data flow or certain event processing [6]. Hence, $C_i \in \{C_i^{typ}, C_i^{cri}\}$. The likelihood of C_i^{cri} occurring is usually lower than that of C_i^{typ} and the critical execution times are generally greater than the typical runtime (i.e., $C_i^{cri} \geq C_i^{typ}$) [6].

A low-priority task may delay a high-priority task if they are using the same shared resource. We represent this blocking delay as $B_i = \max_{\tau_l \in lp(\tau_i)}\{C_l\}$, where $lp(\tau_i)$ is the set of tasks with a priority lower than τ_i. Which jobs of a task τ_i will result in typical or critical execution is not known a priori. The observer task τ_o aims to predict the execution behaviors (i.e., which jobs will contribute to critical invocations in the future).

The leaked information may aid in launching other serious attacks (e.g., overriding or blocking an actuation signal [5]).

We note that the task model considered here is different from that of mixed-criticality systems [7]. Unlike mixed-criticality (or dual-criticality) models, where tasks have criticality levels and lower-criticality tasks are dropped in overloaded situations, we consider an application-driven scenario in which different task instances contribute varying execution times.

B. Threat Model

We consider a noninterference model [8], i.e., the scheduling parameters, the scheduling algorithm, and the resulting schedule are publicly available to the adversary. Besides, all tasks have access to precise clocks (i.e., system timer). We assume that the attacker can compromise a low-priority task (i.e., the observer task, τ_o) to infer the arrival modes of a higher-priority target (i.e., the victim task, τ_v). We do not have any specific assumptions on how the attacker task (τ_o) can be modified to inject inference logic.

III. Leaky Scheduler — A Worst-Case Analysis

Consider a time interval of $D_i = T_i$. In the worst case, τ_i will be preempted (i.e., interfered) $\lceil \frac{T_i}{T_h} \rceil$ times by the jobs of a high-priority task τ_h with inter-arrival time T_h. Hence, total interference delay for τ_i: $I_i = \sum_{\tau_h \in hp(\tau_i)} \lceil \frac{T_i}{T_h} \rceil C_h$. Adding interference and blocking delay with the task's execution time provides the response time (i.e., $R_i = C_i + B_i + I_i$). By assumption, a set of tasks is *schedulable* if $R_i \leq D_i, \forall \tau_i \in \Gamma$.

Considering worst-case analysis, we show that it is feasible to infer non-deterministic task arrival behavior (say, typical and critical jobs) by observing the impact that the low-priority task has on its own response time (i.e., the time between arrival and completion of a task). Based on this simple analysis, we show that there exists a correlation between the victim's execution of typical/critical jobs and the observer's response times. Let us consider the taskset presented in Table I, where there is no blocking delay (i.e., $B_i = 0$ for all three tasks). The victim's typical and critical execution times are 1 unit and 3 units, respectively, and the observer's both typical and critical execution times are 2 units. Depending on the τ_v's job execution modes (typical or critical), the response time of τ_o varies. For instance, when τ_v executes its typical execution job, the response time of τ_o is $[9, 11]$ time units depending on whether τ_x executes typical or critical jobs in the $\lceil \frac{T_o}{T_v} \rceil$ interval. Likewise, when τ_v executes critical execution instances, the response time of τ_o becomes $[15, 17]$ time units depending on τ_x's execution. Thus, depending on the typical or critical job execution of higher-priority tasks, the observer task's response time changes, which, in this example, is in the range of $[9, 17]$ time units. This example suggests that even though the critical execution modes are non-deterministic, a preemptive priority scheduler may *leak* arrival information about a high-priority task (τ_v) to a low-priority one (τ_o).

TABLE I
Example Taskset

Task	Priority	Period	Execution Time	
			Typical	Critical
τ_v	High	10	1	3
τ_x	Medium	15	2	3
τ_o	Low	30	2	2

A. Calculation of Response Times

The analysis we presented above is based on a simple window-based interference calculation that provides an upper bound of the response time. We now provide a tighter bound based on traditional real-time response time analysis [9]. For a given C_i, the response time of τ_i is obtained by the following recurrence relation: $R_i(k+1) = B_i + C_i + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i(k)}{T_j} \right\rceil C_j$. The recurrence starts with $R_i(0) = C_i$. The iteration terminates once the worst-case response time is obtained, i.e., $R_i(k+1) = R_i(k)$ for some value of k or when $R_i(k) > D_i$; in that case, the task is unschedulable.

We modify standard response time calculations for data-driven tasks. Let us define $C_i^{max} = \max(C_i^{typ}, C_i^{cri})$ and $C_i^{min} = \min(C_i^{typ}, C_i^{cri})$. The maximum (minimum) response time R_i^{max} (R_i^{min}) is obtained by replacing C_i in the response time equation with C_i^{max} (C_i^{min}). Depending on which jobs (typical or critical) the tasks execute, the actual response time of τ_i (denoted by R_i^a) varies as follows: $R_i^{min} \leq R_i^a \leq R_i^{max}$.

Following the noninterference rule [8] (see §II-B), an adversary can calculate the ranges of the victim's response times (i.e., $[R_v^{min}, R_v^{max}]$) using offline calculations. However, due to the non-deterministic execution modes, the actual runtime response time R_v^a cannot be inferred a priori. We now present an analysis technique that allows the observer task to infer the future arrivals of victim tasks by measuring the victim's own response times. The observer can extract the differences between its job arrival and completion (i.e., R_o^a) by reading the values from the system clock.

IV. Inferring Typical and Critical Job Arrivals

Recall from earlier discussions (§III) that our inference is based on measuring the response times of the observer. Hence, we want to derive the *likelihood* (i.e., conditional probability) of the runtime of t^{th} job of τ_o (denoted by $\hat{r}_o^{(t)}$) given a past record. Mathematically, this conditional probability is represented as follows: $\mathbb{P}(\hat{r}_o^{(t)} | \hat{r}_o^{(1)}, \hat{r}_o^{(2)}, \cdots, \hat{r}_o^{(t-1)})$. However, for any arbitrary invocation t, we need to keep track of all $t-1$ records from the beginning of system operations. Further, obtaining this conditional probability is challenging in practice, even for a smaller setup.

Note that there are $2^{\lceil \frac{R_o^{max}}{T_v} \rceil}$ different victim-job invocation combinations (typical or critical) between the intervals of consecutive observer's jobs. As a concrete example, assume $T_v = 5$, $T_o = 15$, and let R_o^{max} be some value < 15 which makes the taskset schedulable. Suppose $\lceil \frac{R_o^{max}}{T_v} \rceil = 3$. Hence, there will be 3 job instances of the victim between two consecutive executions of the observer task. Let

τ_v^{typ} and $\tau_v^{c}ri$ denote the typical and critical jobs of the victim task, respectively. The possible combinations include $\{(\tau_v^{typ}, \tau_v^{typ}, \tau_v^{typ}), (\tau_v^{typ}, \tau_v^{typ}, \tau_v^{cri}), \cdots, (\tau_v^{cri}, \tau_v^{cri}, \tau_v^{cri})\}$, i.e., a total of $2^3 = 8$ different invocation patterns which may result in non-unique response times for the observer. Now, including other high-priority tasks, there will be a total of $\prod_{\tau_h \in hp(\tau_o)} 2^{\lceil \frac{R_o^{max}}{T_v} \rceil}$ different invocation patterns. Based on offline response time calculations (§III-A), as we shall see in the paper, a probabilistic inference model in conjunction with a clustering-based classification technique (§IV-A-§IV-C) can predict response times and correlate them with τ_v's job modes (i.e., typical or critical) through runtime observations of τ_o's response times.

As presented next, it is possible to reduce the dimensionality of the conditional probability and *learn* the response time behaviors from a smaller record due to the presence of patterns. For this, we use the probabilistic suffix tree (PST) [10] and clustering technique [11] to isolate typical and critical instances.

Suppose the observer extracts the following traces of response times from the system clock: $r_o^1, r_o^2, r_o^1, r_o^3, r_o^2, r_o^2$, where each r_o^i, $i \in \{1, 2, 3\}$ represents a different response time observation. A PST learns a set of subsequences of different lengths, e.g., $\{r_o^1\}$, $\{r_o^3, r_o^3, r_o^2\}$, each of which can be an indicator of the execution mode of the victim's next job. The proposed PST-based inference enables us to calculate the probability of the victim's arrival and the job type (typical or critical) without having to look back at the entire history, that is, $\mathbb{P}(\hat{r}_o^{(t)} | \hat{r}_o^{(1)}, \cdots, \hat{r}_o^{(t-1)}) \sim \mathbb{P}(\hat{r}_o^{(t)} | \hat{r}_o^{(t-k)}, \cdots, \hat{r}_o^{(t-1)})$, for some predefined starting job k.

A. PST-based Prediction

A PST is a probabilistic model that represents sequences through a tree structure, where each node corresponds to a "suffix" of the sequence, and each node stores the probability distribution of "symbols" (i.e., response times in our context) following that suffix. Constructing a PST involves extracting suffixes, calculating probabilities, and applying thresholds to ensure statistical significance and relevance.

Let $\mathcal{R} = \tilde{r}_1 \tilde{r}_2 \ldots \tilde{r}_\sigma$ be a sequence of response times of length σ, where each symbol $\tilde{r}_i$ belongs to a finite alphabet $\mathcal{A}$. In our setup, the members of the alphabet are in the interval $[R_o^{min}, R_o^{max}]$. We build the PST by extracting suffixes from $\mathcal{R}$ and storing them as nodes in a tree structure. Let us define a suffix $\mathcal{S}_k = \tilde{r}_k \tilde{r}_{k+1} \ldots \tilde{r}_\sigma$ as the substring of $\mathcal{R}$ starting at position k and extending to the end of the sequence. For each k, we consider all possible suffixes of lengths 1 to L, where L is the maximum allowable suffix length.

For each suffix $\mathcal{S}$ of length $|\mathcal{S}| \leq L$, we define the frequency counts as follows: $N(\mathcal{S}) = \sum_{i=1}^{\sigma - |\mathcal{S}|+1} \mathbb{I}(\mathcal{R}_{i:i+|\mathcal{S}|-1} = \mathcal{S})$, where $\mathcal{R}_{i:i+|\mathcal{S}|-1}$ denotes the substring of $\mathcal{R}$ starting at position i and ending at $i + |\mathcal{S}| - 1$, and $\mathbb{I}(\cdot)$ is the indicator function, which equals 1 if the condition is true and 0 otherwise. Let $N(\mathcal{S}\tilde{r}_a)$ denote the count of occurrences where the suffix $\mathcal{S}$ is immediately followed by the symbol $\tilde{r}_a \in \mathcal{A}$: $N(\mathcal{S}\tilde{r}_a) = \sum_{i=1}^{\sigma - |\mathcal{S}|} \mathbb{I}(\mathcal{R}_{i:i+|\mathcal{S}|-1} = \mathcal{S} \text{ and } \mathcal{R}_{i+|\mathcal{S}|} = \tilde{r}_a)$. The conditional probability of observing the response time $\tilde{r}_a$ immediately after the suffix $\mathcal{S}$ is given by: $P(\tilde{r}_a|\mathcal{S}) = \frac{N(\mathcal{S}\tilde{r}_a)}{N(\mathcal{S})}$. This probability is stored at the node corresponding to the suffix $\mathcal{S}$ in the tree.

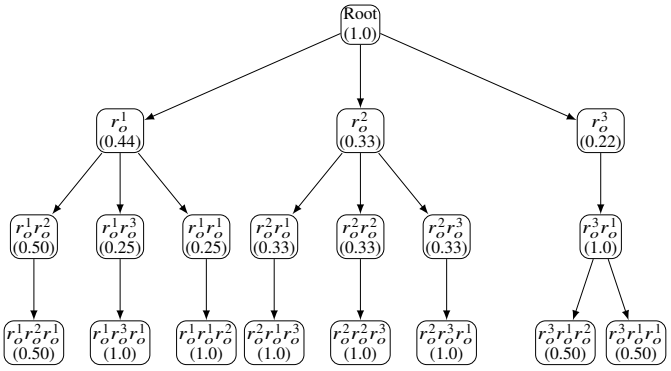


Fig. 1. A PST for a sequence of length 1200 generated with the base pattern of $\mathcal{R} = r_o^1 r_o^2 r_o^1 r_o^3 r_o^1 r_o^2 r_o^2 r_o^3 r_o^1$. The maximum depth is set to 3.

For efficiency reasons, we do not require all suffixes to be included in the tree. For a suffix $\mathcal{S}$ to be included in the tree, each probability $P(\tilde{r}_a|\mathcal{S})$ must satisfy the following condition: $P(\tilde{r}_a|\mathcal{S}) > P_{min}$, where P_{min} is a predefined minimum probability threshold. This criterion ensures that the probabilities are high enough to justify inclusion in the tree. Finally, to update the probabilities, we sort the estimated probabilities $P(\tilde{r}_a|\mathcal{S})$ for each accepted node and then iteratively update the probability distributions for longer suffixes.

Once the PST is constructed, we can predict the future response times from the past sequence. Given a sequence of the observer's response times $\mathcal{R}_o = r_o^1 r_o^2 \ldots r_o^m$, which we refer to as *observation window*, the future response time calculation is based on probability values stored at the deepest matching node, i.e., $\arg\max_{r_o^a \in \mathcal{A}} P(r_o^a|\mathcal{R}_o) = \arg\max_{r_o^a \in \mathcal{A}} \frac{N(\mathcal{R}_o r_o^a)}{N(\mathcal{R}_o)}$.

Example 1. Consider the victim's response time sequence $\mathcal{R} = r_o^1 r_o^2 r_o^1 r_o^3 r_o^1 r_o^2 r_o^2 r_o^3 r_o^1$ repeated 1200 times, where r_o^i indicates the different response time of the observer task τ_o. Figure 1 depicts the resultant PST. Let us focus on the subsequence $r_o^1 r_o^2$. To predict the next response time after $r_o^1 r_o^2$, we examine the occurrences of subsequences following $r_o^1 r_o^2$ in the extended sequence. From this sequence, the possible continuations after $r_o^1 r_o^2$ include $r_o^1 r_o^2 r_o^1$ and $r_o^1 r_o^2 r_o^2$. Counting these occurrences, we find that $r_o^1 r_o^2 r_o^1$ appears 134 times, $r_o^1 r_o^2 r_o^2$ appears 133 times, totaling 267 instances. The probabilities for each continuation are then calculated as follows: $\mathbb{P}(r_o^1|r_o^1 r_o^2 r_o^1) = \frac{134}{267} \approx 0.50$, $\mathbb{P}(r_o^2|r_o^1 r_o^2 r_o^2) = \frac{133}{267} \approx 0.49$. Thus, given the subsequence $r_o^1 r_o^2$, the response time of the next invocation is likely to be r_o^1, r_o^2, each with a probability of approximately 0.50 and 0.49.

B. Inference Algorithm

Algorithm 1 constructs a PST and predicts the future response times from a past sequence. The algorithm begins by initializing the PST with an empty root node. It iterates

Algorithm 1 PST Construction and Inference

1: **Input:** Response times $\mathcal{R} = \tilde{r}_1\tilde{r}_2 \ldots \tilde{r}_\sigma$, suffix length L, threshold P_{min}, training time t_{train}, prediction duration t_{pred}
2: **Output:** Constructed PST and predicted response times
3: **Initialize:** Set the root of the tree as an empty node
4: **for** $k = 1$ to σ **do**
5: Extract suffix $\mathcal{S}_k = \tilde{r}_k\tilde{r}_{k+1} \ldots r_\sigma$
6: **for** $l = 1$ to $\min(L, \sigma - k + 1)$ **do**
7: Define suffix $\mathcal{S} \leftarrow \tilde{r}_k \ldots \tilde{r}_{k+l-1}$
8: Compute frequency count $N(\mathcal{S})$
9: **for** each response time $\tilde{r}_a \in \mathcal{A}$ **do**
10: Compute count of $\mathcal{S}$ followed by $\tilde{r}_a$
11: Calculate $P(\tilde{r}_a|\mathcal{S})$
12: **if** $P(\tilde{r}_a|S) > P_{min}$ **then**
13: Add node $\mathcal{S}$ with probability $P(\tilde{r}_a|S)$
14: **end if**
15: **end for**
16: **end for**
17: **end for**
18: **Prediction:**
19: Given a sequence $\mathcal{R}_v$, identify the longest matching suffix $\mathcal{S}$
20: **for** $time = t_{train}$ to $t_{train} + t_{pred}$ **do**
21: Predict the future arrival of τ_v: $\arg\max_{r_o^a \in \mathcal{A}} P(r_o^a|\mathcal{R}_o)$
22: **end for**

over each position k in the sequence $\mathcal{R}$, extracting suffixes $\mathcal{S}$ starting from each position (Line 5). For each suffix S of length 1 up to the maximum allowable length L, the frequency count $N(\mathcal{S})$ of the suffix within the sequence is computed (Line 8). For each possible response time candidate $\tilde{r}_a$ from the alphabet $\mathcal{A}$, the count $N(\mathcal{S}\tilde{r}_a)$ of occurrences where S is immediately followed by $\tilde{r}_a$ is determined (Line 10). Then we calculate the conditional probability $P(\tilde{r}_a|\mathcal{S})$ of observing $\tilde{r}_a$ after $\mathcal{S}$ (Line 11). If this probability exceeds a predefined threshold P_{min}, the suffix $\mathcal{S}$ is added as a node in the tree with the corresponding probability (Line 13). Finally, the PST uses the longest matching suffix of a given sequence $\mathcal{R}_o = r_o^1 r_o^2 \ldots r_o^m$ to predict the next symbol based on the highest conditional probability (Line 21).

C. Classify Victim's Instances

Recall from our earlier discussion in the beginning of §IV, due to typical and critical invocation patterns of the higher-priority tasks that the observer in an interval of size T_v, there are $\prod_{\tau_h \in hp(\tau_o)} 2^{\lceil \frac{R_o^{max}}{T_v} \rceil}$ different response times for the observer task. As the observer task may observe several response times due to complex arrival patterns of the higher priority tasks, we need to correlate its response times with the actual execution mode of the victim (especially the victim's critical jobs). The PST-based inference presented above helps us to predict the response times of the τ_o based on past observations. Based on this prediction, we adopt a clustering technique to classify which response times of τ_o resulted from the critical (resp. typical) invocation of the victim task. Our reasoning is that, as critical execution modes have longer execution durations in general, when many higher-priority tasks execute their critical jobs, the response times of the observer task will be higher. Hence, it is possible to cluster the response times of τ_o into two buckets and use this information to filter out critical (and typical) jobs of the victim task.

We use K-means clustering [11] to identify a threshold to separate the observer's response times (i.e., which of those caused are by the victim's typical jobs and which are caused by the victim's critical jobs) considering arbitrary execution patterns of other higher-priority tasks between victim and observer (i.e., $\forall \tau_h \in hp(\tau_o) - \{\tau_v\}$). Our intuition is that for a given victim's typical job and irrespective of arbitrary typical/critical instances of other higher-priority tasks $\tau_h \in hp(\tau_o) - \{\tau_v\}$, we put the response times of τ_o into one cluster. Likewise, we do the same for the victim's critical arrivals and isolate the observer's response times in another cluster. At runtime, depending on which cluster the predicted response time maps to (i.e., obtained from Algorithm 1), τ_o can correlate it with the actual typical or critical invocation of the victim task.

D. Timing Considerations

For correct system operation (and to remain stealthy), τ_o should not run more than its worst-case execution time, C_o. Hence, it saves some budget to perform PST-based inference. We define a parameter, λ, whose value is set by the observer to limit the running time of the inference function in each period. This inference duration λ is an integer in the range $0 < \lambda < C_o$. Note that the PST and clustering threshold can be built offline from public knowledge (recall: we follow the non-interference model) and embedded in the task logic (i.e., part of the code binary). We empirically measured the value of λ on a Raspberry Pi platform [12] and found that the inference operations do not have significant overheads (about 50 ms, see §V-B3).

V. Evaluation

A. Demonstration on Cyber-Physical Platforms

To demonstrate the practicality of our analysis and the ways the leaked information can be used by an adversary, we conducted experiments on two cyber-physical platforms, as presented below.

1) Demonstration 1: Manufacturing Robot. We use an off-the-shelf manufacturing robot (known as PiArm, manufactured by SunFounder [13]) as one of our demonstration platforms. Figure 2a shows the robot housed in our lab. The robot is controlled by a Raspberry Pi 4 embedded board. The robot's arm movement is managed by four servos, each connected to a designated `channel` (I/O port) on a daughterboard (Adafruit Motor Shield [14]). We used an open-source, Python-based robot controller [15] to control the robot's movement.

In our setup, we deploy three tasks: a high-priority victim (τ_v), an intermediate task (τ_n), and the observer task (τ_o). During typical operation, the victim task moves the robot's arm and shovel, i.e., it performs the following actions: `move arm down`, `move shovel up`, `move arm up`, `move shovel down`. We consider a scenario where a critical operation requires the robot to move its base. Hence, during the critical mode operation, the victim performs the following actions: `move base left`, `move arm down`, `move shovel up`, `move base right`, `move arm up`, `move shovel down`. Each action takes

(a) (b) (c)

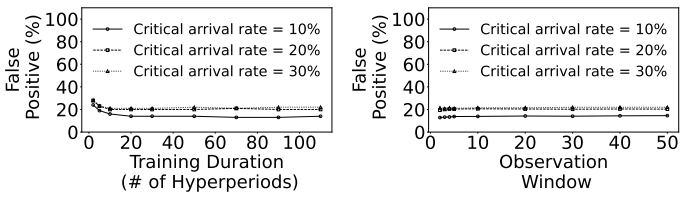
Fig. 2. 3-DoF robot (left) and our custom-built surveillance system: Raspberry Pi with USB camera (middle) and a captured image (right).

a `(channel, angle)` pair that controls the rotation of the corresponding servo to the desired angle. We run the systems over 50 hyperperiods to take response time data for τ_o and use this data to construct our inference model. The observer task τ_o attempts to predict the arrival of the critical mode of τ_v using our learning model. When τ_o predicts the arrival of a critical job from τ_v, it causes the corresponding servo channel to freeze by sending an arbitrary angle value. As a result, the entire robot's operation flow is broken.

Consider this robot is part of a manufacturing assembly chain. Using the proposed inference, an adversary can gauge—with finer precision—when a robot arm is about to pick an object from the conveyor belt. If the robot arm control task is frozen and the robot cannot pick up or move the object from the belt, the entire assembly line may collapse!

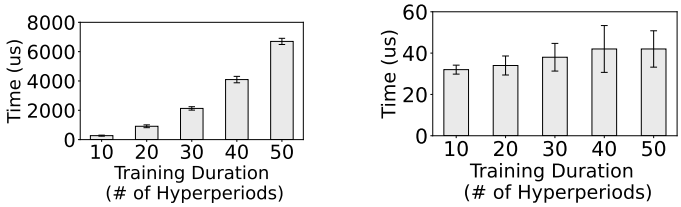
2) Demonstration 2: Surveillance System. Surveillance systems often require continuous monitoring, which leads to high energy and storage consumption. To optimize resource usage, motion-triggered recording can significantly reduce redundant data storage and energy consumption while ensuring that critical events are captured or proper actuations are performed when needed. In this case study, we consider a surveillance system that detects motion in front of an entrance. We built the system using a Raspberry Pi 4 Model B connected to a Logitech USB camera. Figure 2b shows the system setup, and Fig. 2c displays a snap captured by the camera. By default (i.e., in typical mode), the system captures images at coarse granularity, and when a motion is detected (i.e., critical mode), it captures fine-grained, higher-resolution images. To create this setup, we use a video analysis application, Motion [16] (version 4.5.1). Our system comprises five tasks, including the motion detection task. We targeted the motion detection task as our victim task (τ_v). We also created three synthetic tasks (τ_2, τ_3, τ_4) in addition to the observer task (τ_o) running the inference. When no motion is detected, τ_v runs in typical mode, capturing images at 240p in each invocation. Upon detecting a motion near the entrance, τ_v immediately switches to critical mode, records a 10-second video in 720p resolution, and saves it to the filesystem. Like before, we ran the system for 50 hyperperiods to build the model for inference. In this setup, the false positive rate for correctly inferring critical jobs (i.e., when motion was detected) was 16.45%.

Consider this motion-triggered system is coupled with an actuation process (e.g., the door is opened automatically when motion is detected). If an adversary can infer with finer precision when the critical task is about to run (since we have a low false positive rate of 16.45%) and prevent the actuation



(a) (b)

Fig. 3. False positive rates for varying training duration (left) and observation window (right), which remain unaffected.



(a) (b)

Fig. 4. Timing overheads for building the model (left) and performing inference (right). The inference overheads are relatively low (around 50 ms).

(say, opening the door), it may lock people in a facility and disrupt the normal operation of the automation system!

B. Evaluation with Synthetic Tasks

We also use synthetic tasks to check the feasibility of the proposed inference technique.

1) Taskset and Parameters. We vary the system utilization from 0% to 90%. For each system utilization u in the range $[0, 10, \cdots, 90]$%, we generated $N_u = 100$ tasksets, each taskset containing $[7, 20]$ tasks. Task periods were randomly selected from $[100, 900]$ ms. The taskset was generated using the UUniFast algorithm [17]. The tasks follow the rate monotonic (RM) [18] scheduling policy. The tasksets were generated for a fixed hyperperiod of 4500 ms. We did so to ensure fair comparisons among different tasksets for different utilizations. Given the utilization, task period, and fixed hyperperiod, the worst-case execution time $C_i = \max(C_i^{typ}, C_i^{cri})$ was computed. We assumed $C_i^{typ} \leq C_i^{cri}$ and assigned $C_i^{typ} = 0.7C_i^{cri}$. We set the critical arrival rate of tasks between 10% to 30% and varied it as an experimental parameter. Given the arrival rate, the jobs were marked as typical or critical following a uniform distribution. We set $P_{min} = 0.001$ in the experiments.

2) False Positives. Figure 3 presents the false positives. The utilizations ranged from 0% to 90%, and for a training duration of $\hat{H} = 50$ hyperperiods and a runtime history length of past $|\mathcal{R}_o| = 20$ response time observations. False positive numbers show the fraction of instances τ_o mispredicted (i.e., incorrectly classified as critical). As the figures show, the false positive rate for our inference technique is significantly low (less than 25%).

3) Timing Overhead. We conducted our experiments on a Raspberry Pi 4 Model B with 4 GB of RAM. In Fig. 4a, we show the time it takes to build the model (PST + clustering) for different training durations. Likewise, Fig. 4b shows the inference overhead. We repeated the experiments 100 times

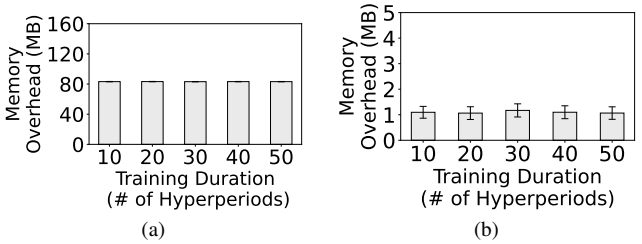


Fig. 5. Memory overheads for building the model (left) and performing inference (right). The memory overhead for runtime inference is low (1 MB).

and reported the 90$^{\text{th}}$ percentile time. As expected, a larger training duration increases training time as the model needs to parse more data to build the tree and form the clusters. Even though we run the experiments on an embedded platform, this cost is offline, and general-purpose computers can also be used (recall: we follow the non-interference model, and the training can be done even before system deployment by simply knowing taskset parameters). The inference timing overheads, as shown in Fig. 4b, are crucial as τ_o needs to make the prediction at runtime from the saved model and by utilizing its past $|\mathcal{R}_o|$ response time measurements. We find that the overheads are relatively low: ~50 us for 50 hyperperiod training duration on the Raspberry Pi board.

4) Memory Overhead. We also carried out similar experiments to measure the memory overhead (see Fig 5). We used Linux's `/usr/bin/time` utility with the `-f "%M"` flag to measure the peak memory usage during τ_o's code execution. As the figures show, the memory footprint remains unchanged with training duration. This is due to the fact that we only need to store a few metadata (e.g., probability values) in the tree and parse it during inference, which is not a memory-hungry operation. Also, we find that runtime memory consumption is very minimal (about 1 MB).

VI. Related Work and Conclusion

Son et al. [19] explore timing covert channels for rate-monotonic schedulers. Volp et al. [8] study covert channels between different priorities of real-time tasks and propose solutions to avoid such covert channels. FrameLeaker [20] allows a pair of tasks to establish a covert channel. There exists other work [21]–[24] on discovering side and covert-channel leakage. Unlike ours, the above studies do not address the prediction of future task arrivals or non-deterministic dual-mode task arrivals. Prior research enforces security restrictions between real-time tasks to prevent sensitive information from leaking via storage channels [25], [26]. While they are proactive defense mechanisms, *(a)* none formally analyze information leakage, and *(b)* they do not work for non-deterministic task models.

To our knowledge, this work is one of the first to explore information leakage in dual-mode real-time systems. Our findings emphasize the need for system designers to examine stealthy leakage in real-time systems and explore new defense mechanisms to mitigate such unwanted information flows.

References

[1] J. P. How, B. Behihke, A. Frank, D. Dale, and J. Vian, "Real-time indoor autonomous vehicle test environment," *IEEE Control Systems Magazine*, vol. 28, no. 2, pp. 51–64, 2008.

[2] N. Raman, R. V. Rachamadugu, and F. B. Talbot, "Real-time scheduling of an automated manufacturing center," *European Journal of Operational Research*, vol. 40, no. 2, pp. 222–242, 1989.

[3] H. Qi, X. Wang, L. M. Tolbert, F. Li, F. Z. Peng, P. Ning, and M. Amin, "A resilient real-time system design for a secure and reconfigurable power grid," *IEEE Transactions on Smart Grid*, vol. 2, no. 4, pp. 770–781, 2011.

[4] K. M. Overmann, D. T. Wu, C. T. Xu, S. S. Bindhu, and L. Barrick, "Real-time locating systems to improve healthcare delivery: A systematic review," *Journal of the American Medical Informatics Association*, vol. 28, no. 6, pp. 1308–1317, 2021.

[5] C.-Y. Chen, S. Mohan, R. Pellizzoni, R. B. Bobba, and N. Kiyavash, "A novel side-channel in real-time schedulers," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 90–102, IEEE, 2019.

[6] S. Quinton, M. Hanke, and R. Ernst, "Formal analysis of sporadic overload in real-time systems," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 515–520, IEEE, 2012.

[7] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, Nov. 2017.

[8] M. Völp, C.-J. Hamann, and H. Härtig, "Avoiding timing channels in fixed-priority schedulers," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pp. 44–55, 2008.

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

[10] G. Bejerano and G. Yona, "Variations on probabilistic suffix trees: statistical modeling and prediction of protein families," *Bioinformatics*, vol. 17, no. 1, pp. 23–43, 2001.

[11] K. P. Sinaga and M.-S. Yang, "Unsupervised k-means clustering algorithm," *IEEE access*, vol. 8, pp. 80716–80727, 2020.

[12] M. Richardson and S. Wallace, *Getting started with Raspberry Pi*. O'Reilly Media, Inc., 2012.

[13] SunFounder, "Piarm documentation." https://docs.sunfounder.com/projects/piarm/en/latest/, 2023.

[14] "Adafruit Motor Shield." https://learn.adafruit.com/adafruit-motor-shield-v2-for-arduino/overview.

[15] SunFounder, "Robot hat." https://github.com/sunfounder/robot-hat, n.d.

[16] Motion Project, "Motion." https://motion-project.github.io/, 2021.

[17] R. I. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *2009 30th IEEE Real-Time Systems Symposium*, pp. 398–409, IEEE, 2009.

[18] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *RTSS*, vol. 89, pp. 166–171, 1989.

[19] J. Son *et al.*, "Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems," in *2006 IEEE Information Assurance Workshop*, pp. 361–368, IEEE, 2006.

[20] M. F. Babar and M. Hasan, "A new covert channel in fixed-priority real-time multiframe tasks," in *2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 1–6, 2024.

[21] M. Völp, B. Engel, C.-J. Hamann, and H. Härtig, "On confidentiality-preserving real-time locking protocols," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 153–162, IEEE, 2013.

[22] X. Gong and N. Kiyavash, "Quantifying the information leakage in timing side channels in deterministic work-conserving schedulers," *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1841–1852, 2015.

[23] J. Kwak and J. Lee, "Covert timing channel design for uniprocessor real-time systems," in *PDCAT*, pp. 159–168, 2019.

[24] A. Ghassami, X. Gong, and N. Kiyavash, "Capacity limit of queueing timing channel in shared fcfs schedulers," in *2015 IEEE International Symposium on Information Theory (ISIT)*, pp. 789–793, 2015.

[25] S. Mohan, M. K. Yoon, R. Pellizzoni, and R. Bobba, "Real-time systems security through scheduler constraints," in *2014 26th Euromicro Conference on Real-Time Systems*, pp. 129–140, IEEE, 2014.

[26] R. Pellizzoni, N. Paryab, M.-K. Yoon, S. Bak, S. Mohan, and R. B. Bobba, "A generalized model for preventing information leakage in hard real-time systems," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 271–282, 2015.