Indistinguishability Prevents Scheduler Side-Channels in Real-Time Systems

Anonymous Author(s)

ABSTRACT

Scheduler side-channels can leak critical information in real-time systems, thus posing serious threats to many safety-critical applications. The main culprit is the inherent determinism in the runtime timing behavior of such systems, *e.g.,* the (expected) periodic behavior of critical tasks. In this paper, we introduce the notion of "schedule indistinguishability", inspired by work in differential privacy, that introduces diversity into the schedules of such systems while offering analyzable security guarantees. We achieve this by adding a sufficiently large (controlled) noise to the task schedules in order to break their deterministic execution patterns. An "ϵ-Scheduler" then implements schedule indistinguishability in real-time Linux. We evaluate our system using two real applications: *(a)* an autonomous rover running on a real hardware platform (Raspberry Pi) and *(b)* a video streaming application that sends data across large geographic distances. Our results show that the ϵ-Scheduler offers better protection against scheduler side-channel attacks in real-time systems while still maintaining good performance and quality-of-service (QoS) requirements.

1 INTRODUCTION

Real-time systems (RTS) have existed for decades in numerous forms, such as avionics systems, nuclear power plants, automobiles, space vehicles, medical devices, power generation and distribution systems as well as industrial robots. Today, however, with the advent of new domains such as autonomous cars, drones, the Internet-of-Things (IoT), and remote monitoring and control, RTS have moved front and center in modern society. Most such systems have *safety-critical* properties, *i.e.,* any problems at run-time could result in significant harm to humans, the system, or even the environment. Imagine a situation in which your car's airbag, a real-time system with stringent timing constraints, *fails to deploy in time*; such a failure can have disastrous results. Despite their importance, security has rarely received adequate attention in the design of real-time cyber-physical systems (CPS). There are many reasons for the lack of robust security: the use of custom hardware/software/protocols, a lack of computing power and memory, and even the notion that such systems lack inherent value to adversaries have limited the development of security mechanisms for them. Since many RTS now use commodity-off-the-shelf (COTS) components and are often connected to each other or even the Internet, they expose additional attack surfaces. In fact, over the past decade, there has been a significant uptick in attacks against cyber-physical systems with real-time properties (*e.g.,* [11, 15, 35, 47, 50, 51, 53, 61]).

RTS have *stringent timing requirements* for ensuring their correct operation. For instance, a typical window for airbag deployment, after a collision is detected, is around 50–60 ms [31] (less than the time it takes to blink!). Such requirements, often driven by the *physical constraints on the system*[1] require that systems be *deterministic at run-time*. Hence, designers take great care to ensure that *(a)* their constituent software tasks execute in an expected manner [39], *e.g.,* to exhibit periodic behavior as shown in Figure 2; *(b)* interrupts are carefully managed [63]; *(c)* memory management is deterministic [37]; and *(d)* running time, on specific processor platforms, is analyzed very carefully at compile/run time (*e.g.,* [9, 12, 26, 59]). However, timing and design constraints further inhibit the addition of security solutions to RTS.

In fact, *the very determinism that is an inherent characteristic of RTS can be used against them as an attack surface*, say, via *timing-based side channels*. Figure 7(a) shows the discrete Fourier transform (DFT) of a real-time system. The graph shows that the deterministic behavior, coupled with the periodic design of RTS, results in a clear demarcation of frequencies (and hence timing behaviors) of critical real-time tasks. This property — that RTS have deterministic behavior — has been used to leak critical information using side channels such as scheduling behavior [14, 52], power consumption traces [33], electromagnetic (EM) emanations [3] and temperature [5]. In particular, ScheduLeak [14], demonstrated *(a)* how to leak timing information from real-time schedules and *(b)* how an adversary can use it to compromise autonomous CPS (*i.e.,* take control of them, or cause them to crash).

Intuitively, one way to reduce determinism (and hence, potentially, increase indistinguishability) in systems is by *adding noise* to system components, for instance, to the schedule. Figures 7(c) and 7(d) show the result of adding Laplacian noise to the system in Figure 7(a). It thereby becomes much more difficult to identify the frequencies of certain tasks because no peaks stand out among the amplitudes. Adding noise to reduce the identification of an individual in a database has been explored in the area of differential privacy [18, 19]. The concept of ϵ-differential privacy is used to measure the confidence with which an individual can be identified in the context of statistical queries in a database. The privacy protection is then quantifiable based on the foundations of mechanisms used to increase the randomness, *e.g.,* drawing noise to be added to the output from, say, the Laplace distributions. Hence, we propose similar ideas to protect RTS by *increasing the indistinguishability of system behaviors*, *e.g.,* the schedule. Hence, at a high level, we propose that:

> *Systems with predictable behaviors are highly susceptible to side-channel attacks; we can protect them by* ***reducing*** *the ability to discern deterministic properties.*

To that end, *we introduce the notion of "ϵ-indistinguishability"* (Section 4) to measure the probability of: information leakage by observation of system behaviors such as schedules and other timing information.

[1] *E.g.,* if a physical component must be actuated at a certain frequency, then some software tasks must also match the rate.

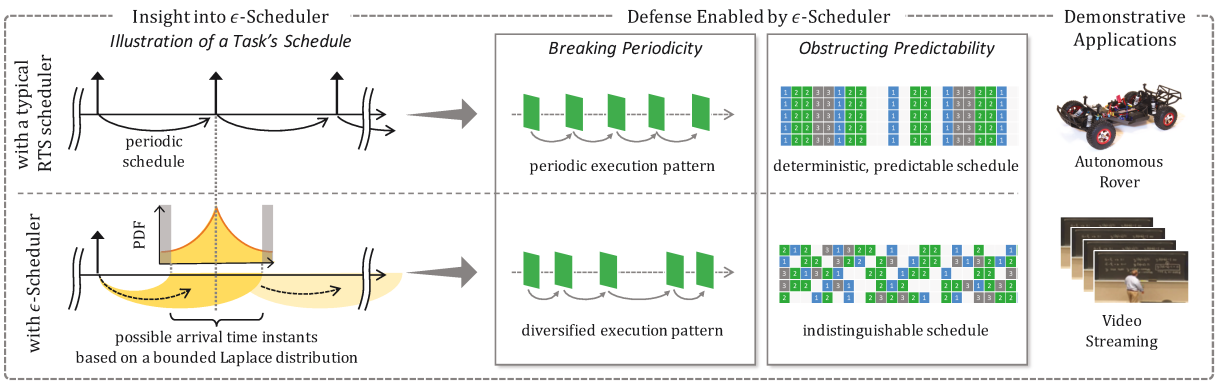


Figure 1: A high-level overview of this paper. The task schedule on the top left depicts a periodic execution pattern (hence predictable and distinguishable) that can be seen in many real-time systems. With the ϵ-Scheduler introduced in this paper, the task schedule is injected with uncertainty (based on Laplace distributions), as depicted on the bottom left. ϵ-Scheduler offers analyzable security and is effective in protecting RTS against scheduler-focused side-channel attacks.

We introduce indistinguishability and noise models in the *resource management algorithms* and, in particular, *schedulers* in real-time CPS. Those components form the core of any RTS and control the precise timing and scheduling behaviors of tasks and resources. Hence, they are the ideal vehicle for *(i)* introducing noise into the system, and *(ii)* measuring the probability of information leakage. We also develop a class of "*ϵ-schedulers*" that incorporate the notion of ϵ-indistinguishability (Section 4). Figure 1 shows an overview of the concepts in this paper using a real world attack from literature.

While some work (*e.g.,* [4, 36, 60]) has proposed the use of *ad hoc* randomization methods in real-time schedulers, their effect is severely restricted since they must adhere to all of the timing constraints in RTS; for instance, these solutions are not allowed to miss even a single deadline. In addition, they do not work well in heavily loaded (*i.e.,* high utilization) systems. This, in conjunction with their *ad hoc* nature, also limits the calculation of any formal security guarantees *w.r.t.* the degree of protection offered. In contrast, our ϵ-schedulers, *(a)* can protect a wider class of RTS, since we propose a modified system model (Section 5.1) that allows for *some* deadlines to be missed, *(b)* can provide formal guarantees (Section 4.2) built off the body of work in differential privacy and *(c)* works on all types of systems, including heavily loaded ones.

The ϵ-Scheduler is implemented on Linux, on both: a hardware platform (Raspberry Pi) running real-time Linux as well as a simulation platform. We evaluate our work using two real applications (an autonomous rover and a video streaming application). We further evaluate the ϵ-Scheduler using simulations to explore the design space as well as potential limitations of our system. The results demonstrate that ϵ-Scheduler is able to not only offer a higher degree of protection (as compared to the state-of-the-art, see Section 8.2), but also do so with actual *guarantees* while still maintaining a high degree of performance and quality-of-service (QoS). In summary, the main contributions of this paper are:

(1) the notion of schedule indistinguishability that captures the difficulty of identifying information about individual tasks in a task schedule [Section 3.3].
(2) an ϵ-Scheduler that implements the schedule indistinguishability concepts based on bounded Laplace distributions [Section 4 and 5].
(3) Implementation on a real hardware platform running real-time Linux that is open-sourced [Section 6].

Note: Our aim is to *modify system states to deter side-channel attacks* and **not** the leakage of private data, the latter being the typical use case for differential privacy.

2 BACKGROUND AND RELATED WORK

2.1 Real-Time Systems and Scheduler Side-Channels

Real-Time Systems. Time-critical systems such as self-driving cars, medicine/vaccine delivery drones, space rovers (*e.g.,* NASA's Opportunity and Spirit), industrial robots, autonomous tractors and unmanned aerial vehicles (UAV), *etc.*, play a vital role in shaping today's technological evolution from everyday living to space exploration. In such systems, tasks[2] delivering critical functionality rely on an operating system (typically an operating system that supports a real-time scheduling policy) to fulfill their timing requirements (*e.g.,* the task must complete within a predefined time limit). Oftentimes, these tasks (*e.g.,* system heartbeat keepers, PID control processes, sensor data collectors, motor actuators, *etc.*) are designed to execute in a *periodic* fashion to guarantee responsiveness. Such real-time tasks are usually associated with a set of predefined timing constraints such as *(a)* minimum inter-arrival times (*i.e.,* periods), *(b)* deadlines and *(c)* worst-case execution times (WCET). They are scheduled using well-known real-time scheduling algorithms *e.g.,* fixed-priority preemptive scheduling, earliest deadline first scheduling [39]. These real-time constraints help system designers ensure that all safety guarantees are met (*e.g.,* no real-time tasks will miss their deadlines). As a result, the system schedule becomes deterministic and highly predicable.

Scheduler Side-Channels. The aforementioned determinism and predictability, though favorable for the system safety, is a double-edged sword – they create side-channels in RTS. There has been an increasing focus (*e.g.,* [22, 23, 34, 52, 54, 55, 57, 62]) on studying and demonstrating the existence of side-channels and covert-channels (as consequences of the determinism) in RTS. In this paper, we are

[2]A task in typical real-time systems corresponds to a process/thread in generic operating systems. In this paper, we will use "task" and "process" interchangeably.

particularly interested in the *side-channels that leak system timing behavior via task schedules*. In the RTS domain, Chen *et al.* [14] first introduced the scheduler side-channels using the ScheduLeak algorithms. They extract execution behavior of critical real-time tasks from an observed task schedule at run-time. Liu *et al.* [42] used the same attack surface (*i.e.,* the task schedule) and showed that precise timing values of critical real-time tasks can be uncovered using frequency spectrum analysis (*e.g.,* Discrete Fourier Transform, DFT, analysis) as shown in Figure 7. Such timing information, while seemingly subtle, is a crucial stepping stone to launching many attacks against RTS. Consequently, additional side-channels such as power consumption traces [33], schedule preemptions [14, 52], electromagnetic (EM) emanations [3] and temperature [5] have been demonstrated in RTS. In Chen *et al.* [14] have also shown how such information leakage can be used to launch more deliberating attacks, *e.g.,* taking control of autonomous systems.

Schedule Obfuscation. Yoon *et al.* [60] attempted to tackle the scheduler side-channels by introducing a randomized scheduling algorithm that obfuscates the task schedules in fixed-priority preemptive RTS. This idea has been extended to multi-core environments [4]. Similarly, Krüger *et al.* [36] developed a combined online/offline randomization scheme to reduce determinisms for time-triggered systems. Nasri *et al.* [48] conducted a comprehensive study on the schedule randomization approach and argued that such techniques can actually expose the fixed-priority preemptive RTS to more risks. Burow *et al.* [10] explore several moving-target defenses (randomization-based) against different types of attacks in the context of RTS (including soft RTS). While this existing work is centered on the problem of scheduler side-channels, they do not provide analytical guarantees for the protection against such attacks. Additionally, the work targets highly constrained real-time systems and hence their effectiveness is often limited. In contrast, we focus on a more realistic RTS model that has flexible and more tolerable timing requirements. This enables us to explore a more aggressive defense strategy to achieve higher (and analyzable) protection against the threats imposed by scheduler side-channels.

2.2 Differential Privacy and Randomized Mechanisms

Differential Privacy. Differential privacy, along with the theorems and algorithms that build the foundation for protecting data privacy, was originally introduced [18, 19] in the context of statistical queries on databases. It can be seen that differential privacy is used in many subjects addressing the issue of data privacy [13, 18]. There is also a growing trend to extend such concepts to the systems domain [17, 30, 58] to protect data privacy distributed among a group of devices. While in this paper *we focus on the system security rather than data privacy*, the high-level goal is somewhat similar to differential privacy and hence relevant techniques may be adopted.

In our context, we define the notion of *task/job indistinguishability* that defines the probability of distinguishing the execution states of one task/job from another in task schedules. Roughly speaking, a low indistinguishability enables an adversary to identify a task's execution from an observed schedule with a high confidence and hence the system is prone to compromises via scheduler side-channels. To address such a problem, we propose an ϵ-Scheduler that offers "ϵ-indistinguishability" at a job level and/or a task level, subject to system constraints as well as the system designer's security goal. To the best of our knowledge this paper is the first work that adopts the foundation of differential privacy in the design of schedulers and especially to address the security issues in RTS.

Laplace Mechanism. The Laplace distribution has been used in the classic differential privacy problems for generating random noise to achieve desired privacy protections [19]. Conventionally, the Laplace distribution has a probability density function defined as $\text{Lap}(x \mid \mu, b) = \frac{1}{2b}\exp(-\frac{|x-\mu|}{b})$. In this paper, we use the Laplace distribution to generate randomized inter-arrival times for each job at run-time. While there can be random noise drawn from other distributions (*e.g.,* Gaussian distribution [28, 41]) achieving the same level of indistinguishability using the Laplace distribution allows us to reuse existing mathematical and algorithmic components with the theoretical foundations from the differential privacy domain.

3 SYSTEM AND ADVERSARY MODELS

3.1 Preliminaries

The sets of natural numbers and real numbers are denoted by $\mathbb{N}$ and $\mathbb{R}$. For a given $n \in \mathbb{N}$, the set $[n]$ represents $\{1, 2, ..., n\}$. We denote the Laplace distribution with location μ and scale b and $\text{Lap}(b)$ by $\text{Lap}(\mu, b)$ and we write $\text{Lap}(b)$ when $\mu = 0$. For a random variable x, drawing values from a Laplace distribution is denoted by $x \sim \text{Lap}(\cdot)$. As conventionally used, we sometimes abuse notation and denote a random variable $x \sim \text{Lap}(\cdot)$ simply by $\text{Lap}(\cdot)$.

We consider a discrete time model [32]. In our context, we mainly focus on the issue that is concerned with the timing in a single node real-time system. We assume that a unit of time equals a timer tick governed by the operating system and the corresponding tick count is an integer. That is, all system and real-time task parameters are multiples of a time tick. We denote an interval starting from time point a and ending at time point b that has a length of $b - a$ by $[a, b)$ or $[a, b - 1]$.

3.2 Real-Time System Model

In this paper, we consider a single processor, preemptive real-time system in which some deadline misses are tolerable [16, 43]. Such systems are very common, *e.g.,* the system contains a set of N real-time tasks $\Gamma = \{\tau_i \mid i \in [N]\}$, scheduled by a dynamic-priority scheduler (*e.g.,* Earliest Deadline First, EDF, scheduler [39]). We assume the real-time tasks are independent (*i.e.,* no dependencies between tasks). A real-time task can be a periodic task (with a fixed period) or a flexible task (that has flexible period choices within a predefined range)[3] [44]. We model a real-time task τ_i by a tuple $(\mathcal{T}_i, \mathcal{D}_i, C_i, \eta_i)$ where $\mathcal{T}_i = \{T_{i,k} \mid k \in \mathbb{N}\}$ is a set of admissible periods, $\mathcal{D}_i = \{D_{i,k} \mid k \in \mathbb{N}\}$ is a set of implicit, relative deadlines (*i.e.,* $D_{i,k} = T_{i,k}, \forall k \in \mathbb{N}$), C_i is the worst-case execution time (WCET) and η_i is a *task inter-arrival time function* as defined below (a glossary table is provided in Appendix Table 4 for reference). It can be easily seen that a periodic task is then a flexible task where the "choice of periods" is limited to a single value. That is, $\mathcal{T}_i = \{T_{i,1}\}$

[3]The system can also contain other sporadic and aperiodic tasks. Yet, these types of tasks do not naturally demonstrate periodicity by design and thus are not of interest in our context. For this reason, we intentionally exclude these types of tasks in our task model to be focused on the periodic components.

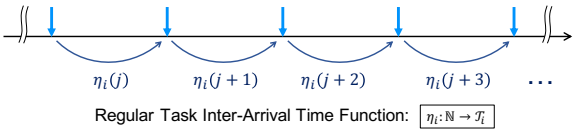


Figure 2: Illustration of the task execution model used in this paper. Arrows represent the scheduled arrival time instants. The distance between two adjacent arrival times of a task is modeled by a task-specific function η_i.

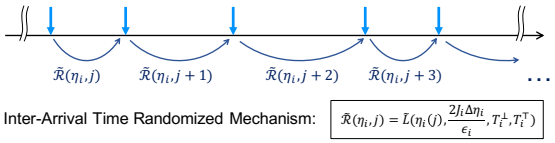


Figure 3: Illustration of the task execution after injecting noise. The inter-arrival times become irregular and unpredictable with using a randomized mechanism.

when τ_i is a periodic task and we sometimes use T_i to denote such a fixed period for simplicity. A task's execution instance is aborted upon missing its current deadline and it does not impact the release of the task's next execution instance.

To formulate the problem better, let us assume that a task's execution behavior is modeled by a *task inter-arrival time function* where each task has a dedicated function, as illustrated in Figure 2.

Definition 3.1. (Task Inter-Arrival Time Function.) For a task τ_i the inter-arrival time function is defined as

$$\eta_i : \mathbb{N} \to \mathcal{T}_i \tag{1}$$

where $\eta_i(j)$ is the task's inter-arrival time at the j^{th} instance. The resulting inter-arrival time is one of the values in the task's inter-arrival time set, $\eta_i(j) \in \mathcal{T}_i$. ■

Note that a strict periodic task (*i.e.,* $\mathcal{T}_i = \{T_{i,1}\}$) always gets a fixed output from its inter-arrival time function, $\eta_i(j) = T_{i,1}, \forall j \in \mathbb{N}$. Then, based on the above function, the system's timing behavior (*w.r.t.* the task deadlines and inter-arrival times) can be modeled by $\eta_i, \forall \tau_i \in \Gamma$. That is, when the j^{th} instance of task τ_i arrives, the scheduler computes its period from $\eta_i(j)$ and configures its deadline as well as the next arrival time accordingly.

3.3 Adversary Model

We are mainly concerned about scheduler side-channels that are exposed by the deterministic nature of RTS as introduced in Section 2. We assume that an adversary observes the system schedule via some existing side-channels [3, 5, 14, 33, 52]. We further assume that the adversary does not have access to the scheduler. Without this assumption, the adversary can undermine the scheduler or directly obtain the schedule information without using the side-channels.

Note that some existing attacks have demonstrated that periodicity can be exploited to learn a targeted task's execution state that can be used to launch further, more critical attacks on the system with higher precision [14, 42]. These types of attacks rely on the fact that periodicity exists in the real-time tasks being targeted. In this paper, we aim to eliminate such scheduler side-channels by obscuring the task periodicity in the schedule. To this end, our goal in this paper is to achieve *schedule indistinguishability* in the system that can be further categorized into:

(i) Job-level indistinguishability refers to the difficulty of distinguishing a task's job from another of the *same task* in a task schedule. As introduced in Section 3.2, a flexible task can have multiple predefined periods that are associated to different execution modes and purposes. For instance, a feedback control task in a cyber-physical system can adjust its period based on the severity of error the physical asset under control is experiencing [44]. Leaking the current period of the control task reveals the system's internal state as well as the physical asset's external state. Achieving a job-level indistinguishability for such a task weakens the adversary's ability to reason about the task's internal execution state.

(ii) Task-level indistinguishability, on the other hand, refers to the difficulty of distinguishing a task from another in a schedule. In a RTS in which all tasks are strictly periodic, it is generally not hard to distinguish and identify individual tasks from a schedule (see Section 8.2.1 for an example and analysis). As a result, tasks are at risk of leaking critical information. For instance, in the ScheduLeak attack [14], the adversary exploits the periodicity to extract the execution behavior of a critical real-time task. Achieving task-level indistinguishability weakens the adversary's ability to glean information about a specific task from the schedule.

It's intuitive to see that job-level indistinguishability is a necessary condition for the task-level indistinguishability. That is, if task-level indistinguishability can be achieved, then job-level indistinguishability is also achievable. It's worth pointing out that the inverse relation does not hold: achieving individual job-level indistinguishability does not automatically grant the task-level indistinguishability. Yet, in practice, there exist real-time constraints that restrict the degree of timing that we can tweak. In such cases, the task-level indistinguishability may be infeasible to achieve. In this paper, we propose an extended task model and a real-time scheduler with an inter-arrival time randomized mechanism to achieve job-level indistinguishability and, when feasible, task-level indistinguishability.

4 SCHEDULE INDISTINGUISHABILITY

In this section we introduce the components (inter-arrival time sensitivity and randomized mechanism) that achieve notions of the job/task-level indistinguishability. These are fundamental to developing the ϵ-Scheduler that will be introduced in Section 5.

4.1 Randomizing Inter-Arrival Times

Let's consider a task τ_i and its inter-arrival time function η_i. The function produces consistent inter-arrival times. To break this predictable behavior, we intend to *randomize each inter-arrival time.* To this end, we propose an *inter-arrival time randomized mechanism*, denoted by $\mathcal{R}(\cdot)$, that is attached to the scheduler to add random noise. The inter-arrival time randomized mechanism is defined as:

$$\mathcal{R}(\tau_i, j) = \lfloor \underbrace{\eta_i(j)}_{\text{the } j^{th} \text{ inter-arrival time of the task } \tau_i} + \underbrace{Y}_{\text{random noise drawn from some distribution centered at 0}} \rceil \tag{2}$$

where $\tau_i \in \Gamma$, $j \in \mathbb{N}$ represent the j^{th} inter-arrival time of the task τ_i. Y is a random noise value drawn from some distribution centered at 0. Note that the noise Y is presented separately for the purpose of

illustration. Such a representation is the same as drawing a random value from some distribution centered at $\eta_i(j)$ – which is what the ϵ-Scheduler is eventually based on. The outcome is rounded to the nearest integer and taken as the randomized inter-arrival time.

The added random noise Y creates inconsistent inter-arrival times for a task and breaks a task's periodicity. Yet, without specifying a noise distribution, it may be insufficient to obscure the task's behavior, for example, when the noise's variance is insignificant. Before examining the noise addition mechanism, we first formally define indistinguishability in our context.

4.2 Inter-Arrival Time Indistinguishability

As introduced in Section 3.3, we are concerned with job/task-level indistinguishabilities. To analyze such indistinguishabilities with the mechanism defined in Equation 2, we use a concept that's similar to the notion of differential privacy [18, 19].

Definition 4.1. (ϵ-Indistinguishability Inter-Arrival Time Randomized Mechanism.) An inter-arrival time randomized mechanism $\mathcal{R}(\cdot)$ is ϵ-indistinguishable if

$$\Pr[\underbrace{\mathcal{R}(\tau, j)}_{\text{any randomized inter-arrival time for any given task } \tau} \in \mathcal{S}] \leq e^{\epsilon} \Pr[\underbrace{\mathcal{R}(\tau', j')}_{\text{any randomized inter-arrival time of any given task } \tau'} \in \mathcal{S}] \tag{3}$$

for all $\tau, \tau' \in \Gamma$, $j, j' \in \mathbb{N}$ and $\mathcal{S} \subseteq \text{Range}(\mathcal{R})$. ■

That is, $\mathcal{R}(\cdot)$ enables inter-arrival time indistinguishability for a single job instance if Equation 3 is satisfied.

Note that Definition 4.1 is general enough to consider both the job-level and task-level indistinguishabilities. When $\tau \neq \tau'$, task-level indistinguishability is implied; when $\tau = \tau'$, job-level indistinguishability is implied. It is worth noting that we can maintain an independent ϵ_i value for each task τ_i and each of them achieves their own ϵ_i-indistinguishability. The indistinguishability for the whole task set is determined by the worst of the ϵ_i values [46] (that corresponds to the task-level indistinguishability).

4.3 Inter-Arrival Time Sensitivity and Noise

To determine the degree of noise to be added to make two inter-arrival times indistinguishable, We define "*inter-arrival time sensitivity*". Intuitively, the value of the inter-arrival time sensitivity is assigned by the largest possible difference between two inter-arrival times. However, the true assignment depends on the protection goal (*i.e.,* whether to achieve the job-level indistinguishability or the job-level indistinguishability), as explained below.

Definition 4.2. (Inter-Arrival Time Sensitivity.) This reflects the sensitivity of the function $\eta_\tau(\cdot)$ defined, depending on the desired indistinguishability goal, as:
(i) Job-level indistinguishability: the inter-arrival time sensitivity for the job-level indistinguishability, denoted by $\Delta\eta_\tau$, for a given task τ, is defined as

$$\Delta\eta_\tau =: \max_{\substack{j,j' \in \mathbb{N} \\ j \neq j'}} | \underbrace{\eta_\tau(j) - \eta_\tau(j')}_{\text{distance between any two inter-arrival times of the task } \tau} | \tag{4}$$

that is task-specific.
(ii) Task-level indistinguishability: the inter-arrival time sensitivity, denoted by $\Delta\eta_\Gamma$, is defined as:

$$\Delta\eta_\Gamma =: \max_{\substack{\tau,\tau' \in \Gamma \\ j,j' \in \mathbb{N}}} | \underbrace{\eta_\tau(j) - \eta_{\tau'}(j')}_{\text{distance between any two inter-arrival times of any two tasks in the task set } \Gamma} | \tag{5}$$

that is task-set-dependent. ■

For simplicity, we use $\Delta\eta$ to represent either of the sensitivities when the context is clear. Then, the use of the Laplace distribution $\text{Lap}(\eta_\tau, \frac{\Delta\eta}{\epsilon})$ for generating the randomized inter-arrival times preserves the ϵ-indistinguishability from Definition 4.1 for a single job instance. This property can be easily proved by expanding Equation 3 with the probability density function of the $\text{Lap}(\eta_\tau, \frac{\Delta\eta}{\epsilon})$ distribution [19, Theorem 3.6]. Therefore, the job-level indistinguishability is achieved when $\Delta\eta = \Delta\eta_\tau$ and the task-level indistinguishability can be achieved when $\Delta\eta = \Delta\eta_\Gamma$.

4.4 ϵ-Indistinguishability in J Instances

The randomized mechanism $\mathcal{R}(\cdot)$ with Laplace noise $\text{Lap}(\frac{\Delta\eta}{\epsilon})$ offers ϵ-indistinguishability for a single instance. However, an attacker typically observes a longer sequence from the schedule. Therefore, we are more interested in the conditions for achieving ϵ-indistinguishability for a certain duration (as opposed to a single point in time). As a noise draw occurs for every job instance, based on the theorem of Sequential Composition [46, Theorem 3], the privacy degradation is cumulative as the number of draws increases. A smart attacker may be able to sort out the distribution by collecting sufficient samples. Therefore, it is crucial to understand the condition for providing the required level of indistinguishability for a certain duration. To this end, we measure the duration in the number of job instances (that corresponds to the number of noise draws for the corresponding inter-arrival times). Then we use the following theorem to determine the scale of the noise for preserving ϵ-indistinguishability up to J job instances.

Theorem 4.3. *The Laplace randomized mechanism $\mathcal{R}(\cdot)$ with the scale $\frac{J\Delta\eta}{\epsilon}$ is ϵ-indistinguishable up to J job instances.* ■

This theorem can be proved by expanding Equation 3 with J invocations of $\mathcal{R}(\cdot)$. The proof is given in Appendix A for reference. The assignment of J for a given task set is discussed in Section 5.3.

4.5 Bounded Laplace Randomized Mechanism

While the introduced Laplace randomized mechanism offers ϵ-indistinguishability, the unbounded output domain for the randomized inter-arrival times makes it infeasible to adopt in real systems. To address this problem, we introduce the "bounded Laplace randomized mechanism", *i.e.,* the randomized inter-arrival time drawn from a Laplace distribution is bounded by a given range. There are typically two solutions for bounding the value drawn from a distribution: *(i)* truncation and *(ii)* bounding [40]. Truncation projects values outside the domain to the closest value within the domain. Bounding, used in this paper, is to continue sampling independently from the distribution until a value within the specified range

is returned. Let's denote such a bounded Laplace distribution by $\widetilde{L}(\mu, b, T^{\perp}, T^{\top})$ of which the drawn value is in the range $[T^{\perp}, T^{\top}]$.

Using such a bounded Laplace distribution allows a mechanism to return randomized inter-arrival times within a range that's feasible for the given constraints. However, it is known that the bounded Laplace distribution cannot preserve the same level of probabilistic guarantee (*i.e.,* the ϵ-indistinguishability in our context) with the same scale parameter as a pure Laplace distribution and a doubling of the noise variance is required to compensate for the loss [29, 40]. Based on this condition and Theorem 4.3, we define the bounded inter-arrival time Laplace randomized mechanism as follows:

Definition 4.4. (Bounded Inter-Arrival Time Laplace Randomized Mechanism.) Let $[T_i^{\perp}, T_i^{\top}]$ be the feasible inter-arrival time range for a given task τ_i, the bounded inter-arrival time Laplace randomized mechanism is defined as

$$\widetilde{\mathcal{R}}(\tau_i, j) = \widetilde{L}(\eta_i(j), \frac{2J_i\Delta\eta_i}{\epsilon_i}, T_i^{\perp}, T_i^{\top}) \quad (6)$$

(j^{th} inter-arrival time of τ_i; scale of the noise distribution; bounds for randomized inter-arrival time)

where $\widetilde{L}(\cdot)$ is the bounded Laplace distribution of which the drawn values are bounded in the range $[T_i^{\perp}, T_i^{\top}]$ based on a pure Laplace distribution $\text{Lap}(\eta_i(j), \frac{2J_i\Delta\eta_i}{\epsilon_i})$. ■

The variables $T_{\perp}$, $T_{\top}$, $\Delta\eta_i$, J_i and ϵ_i are extended task parameters of τ_i to be formalized in Section 5.1. Following Theorem 4.3, the bounded inter-arrival time Laplace randomized mechanism $\widetilde{\mathcal{R}}(\tau_i, j)$ is ϵ-indistinguishable up to J job instances.

5 ϵ-SCHEDULER

With the components described in Section 4, we now introduce our proposed real-time scheduler, the ϵ-Scheduler. In each task's arrival (the beginning of a new instance), the ϵ-Scheduler uses $\widetilde{\mathcal{R}}(\cdot)$ for generating the task's next arrival time (*i.e.,* randomizing inter-arrival times). In this section we first introduce an extended RTS task model that supports such an ϵ-Scheduler, followed by discussion for how the extended task parameters can be determined for a given system to achieve job/task-level indistinguishability.

5.1 Extended Task Model

The basic task model presented in Section 3.2 is extended to include parameters necessary for an ϵ-Scheduler to achieve the desired indistinguishability. In ϵ-Scheduler, a task τ_i is characterized by $(\mathcal{T}_i, \mathcal{D}_i, C_i, \eta_i, T_i^{\perp}, T_i^{\top}, \Delta\eta_i, J_i, \epsilon_i)$ where $[T_i^{\perp}, T_i^{\top}]$ is a range of tolerable periods, $\Delta\eta_i \geq 0$ is the inter-arrival time sensitivity parameter, J_i is the task's effective protection duration, and $\epsilon_i > 0$ is the indistinguishability scale parameter. At each new job arrival, the ϵ-Scheduler invokes $\widetilde{\mathcal{R}}(\tau_i, j) = \widetilde{L}(\eta_i(j), \frac{2J_i\Delta\eta_i}{\epsilon_i}, T_i^{\perp}, T_i^{\top})$ to determine the next job's randomized arrival time point.

In this extended task model, the parameters $\mathcal{T}_i, \mathcal{D}_i, C_i, \eta_i, T_i^{\perp}$ and $T_i^{\top}$ are obtained from the system dynamics. The additional parameters $\Delta\eta_i$, J_i and ϵ_i are to be given by the system designer. As the degree of noise added to a task's inter-arrival time relies on the extended parameters, it is crucial to assign proper values based on the desired indistinguishability goal. We now discuss the considerations for determining these values.

5.2 Determining Inter-Arrival Time Sensitivity

$\Delta\eta_i$ represents *the degree of random noise needed to make two inter-arrival times indistinguishable* and can be determined based on Definition 4.2. The value of $\Delta\eta_i$ should be fixed for an execution instance once assigned. In the case that we intend to achieve job-level indistinguishability is to achieve for a given task τ_i, the value of $\Delta\eta_i$ is determined solely by the task's set of periods, $\mathcal{T}_i$. In this case, each task's sensitivity is independent of each other. On the other hand, task-level indistinguishability requires that the sensitivity reflects all tasks in the system. Hence, the sensitivity for the task-level indistinguishability is task set specific and all tasks are assigned with the same sensitivity value. It is straightforward to see that task-level sensitivity will be greater than job-level sensitivity of any task (and hence larger noise will be added). It is up the system designer to decide, taking potential performance degradation into account, which type of indistinguishability should be achieved.

5.3 Calculating Protection Duration

Using the bounded Laplace mechanism, $\widetilde{\mathcal{R}}(\cdot)$, an ϵ-Scheduler is able to preserve ϵ_i-indistinguishability up to J_i job instances for a given task. As pointed out in Section 4.4, the more noise samples collected, the more likely an attacker is able to reconstruct the distribution and reveal a task's behavior. Therefore, ϵ_i-indistinguishability can't be guaranteed for an infinite time. For this reason, the ϵ-Scheduler should be used with other security measures for comprehensive protection against scheduler side-channels. There exist some security schemes that work well together in this context. For instance, one can perform periodic security checks to detect possible intrusions and anomalies [27]. With such a scheme, the distance between two security checks can be used as a reference to compute the protection duration parameter J_i. Another feasible scheme is the restart-based mechanism [1, 2] that enforces a reboot once a while. In such a case, the maximum time to reset the system can be used to compute J_i. In both schemes, the adversary's attack progress is disrupted once the corresponding security measure kicks in and the ϵ-Scheduler offers further security guarantees from compromise via scheduler side-channels. Note that J_i is defined in the *number of job instances* as each job arrival draws a random value from the distribution. When the job-level indistinguishability is considered, each task's J_i is computed independently so the value can be different across tasks. Let λ be the protection duration in time, then

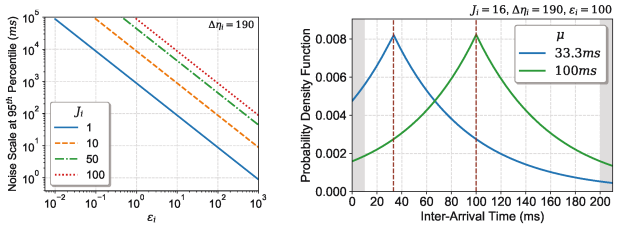
$$J_i = \left\lceil \frac{\lambda}{\min(\mathcal{T}_i)} \right\rceil \quad (7) \qquad J_i = \max\left(\left\lceil \frac{\lambda}{\min(\mathcal{T}_j)} \right\rceil \mid \tau_j \in \Gamma \right) \quad (8)$$

(desired protection duration; the smallest period of τ_i; the smallest period in the task set Γ)

Equation 7 offers ϵ_i-indistinguishability to τ_i within λ time. For task-level indistinguishability, J_i for all tasks must be equal to offer the desired indistinguishability guarantee (subject to ϵ_i) as calculated by Equation 8 where λ is a *global protection duration* in time.

5.4 Choosing Indistinguishability Parameter

With the noise level ($\Delta\eta_i$) and protection duration (J_i) determined for a given task set, ϵ_i is the major remaining variable that a system



(a) Noise scale of $\text{Lap}(0, \frac{2J_i\Delta\eta_i}{\epsilon_i})$ **(b) Probability density**

Figure 4: (a) The noise scale of $\text{Lap}(0, \frac{2J_i\Delta\eta_i}{\epsilon_i})$ **at** 95^{th} **percentile with** $\Delta\eta_i = 190ms$ **and varying** ϵ_i **and** J_i**. Both axes are displayed in a base 10 logarithmic scale. (b) Probability density of the randomized inter-arrival times for the task** τ_i **with** $\mathcal{T}_i = \{33.33ms, 100ms\}$**. The blue and green lines show the distribution when the desired period is at** $33.33ms$ **and** $100ms$ **respectively. In this case,** ϵ**-Scheduler offers a job-level** ϵ**-indistinguishability for** τ_i **with** $\epsilon_i = 100$**,** $\Delta\eta_i = 190$ **and** $J_i = 16$**.**

designer must specify to secure the desired degree of protection. Ideally, a smaller ϵ_i value provides a better indistinguishability by generating randomized inter-arrival times with larger noise scale. However, a large noise scale may sometimes be impractical for real-time applications. Figure 4(a) shows examples of noise scales (the y-axis, represented by the 95^{th} percentile) with varied ϵ_i values (the x-axis) for a fixed $\Delta\eta_i = 190ms$ and various J_i settings. It suggests that an ϵ_i value above an order of magnitude can be practical to most RTS. Figure 4(b) shows an example of the distributions of the inter-arrival times for a task that has $\mathcal{T}_i = \{33.33ms, 100ms\}$ with $\Delta\eta_i = 190ms$, $J_i = 16$ (with assuming $\lambda = 500ms$) and $\epsilon_i = 100$. It shows how a task's inter-arrival times are randomly generated by an ϵ-Scheduler in a typical RTS setting.

Nevertheless, a suitable value for ϵ_i is highly system-dependent. Ultimately, it is up to the system designer to select a value based on the overall security and performance goals. Note that all tasks must be assigned an identical ϵ value to achieve task indistinguishability while each task can have an independent ϵ value for job indistinguishability.

6 IMPLEMENTATION IN LINUX

We implemented ϵ-Scheduler in both *(a)* real-time Linux kernel running on Raspberry Pi and *(b)* an open-source[4] simulation platform that we developed. The simulation is used for design space exploration (Section 8) and the real-time Linux kernel is used for demonstration with real hardware and applications and also to analyze overheads. In this section we provide the platform information (also summarized in Table 5 in Appendix) and an overview of the implementation in the real-time Linux kernel.

6.1 Platform and Operating System

We used a Raspberry Pi 4 (RPi4) Model B[5] development board as the base platform for our implementation. RPi4 runs a vendor-supported open-source operating system, *Raspbian* (a variant of Debian Linux). We forked the Raspbian kernel and modified it to implement the proposed ϵ-Scheduler. Since we focus on the single core environment in this paper, the multi-core functionality of RPi4 was deactivated by disabling the `CONFIG_SMP` flag during the Linux kernel compilation phase. The boot command file was also set with `maxcpus = 1` to further ensure the single core usage.

Real-time Environment. The mainline Linux kernel does not provide any hard real-time guarantees even with the custom scheduling policies (*e.g.,* `SCHED_FIFO`, `SCHED_RR`, `SCHED_DEADLINE`). However the *Real-Time Linux (RTL) Collaborative Project*[6] maintains a kernel (based on the mainline Linux kernel) for real-time purposes. This patched kernel (known as the PREEMPT_RT) ensures real-time behavior by making the scheduler fully preemptable. In this paper, we use a PREEMPT_RT-patched kernel (4.19.71-rt24+) to enable real-time functionality. To further enable the fully preemptive functionality for the PREEMPT_RT patch, the `CONFIG_PREEMPT_RT_FULL` flag was enabled during the kernel compilation phase. Furthermore, the variable `sched_rt_runtime_us` was set to -1 to disable the throttling of the real-time scheduler. This setting allows the real-time tasks to use up the entire 100% CPU utilization if required[7]. Also, the active core's `scaling_governor` was set to `performance` mode to disable dynamic frequency scaling during the experiments.

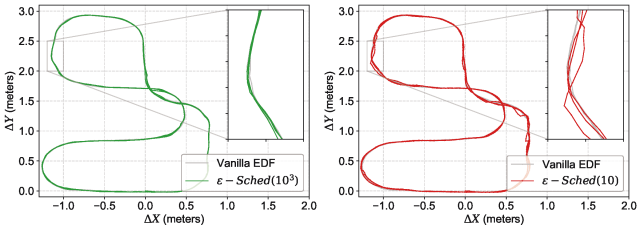
Vanilla EDF Scheduler. Since Linux kernel version 3.14, an EDF implementation (*i.e.,* `SCHED_DEADLINE`) is available in the kernel code base [21]. Therefore, we used this built-in scheduler as the baseline EDF implementation and extended it to implement an ϵ-Scheduler. In Linux the system call `sched_setattr()` is invoked to configure the scheduling policy for a given process. By design, the EDF scheduler in Linux has the highest priority among all the supported scheduling policies (*e.g.,* `SCHED_NORMAL`, `SCHED_FIFO` and `SCHED_RR`). It's also worth noting that the Linux kernel maintains a separate run queue for `SCHED_DEADLINE` (*i.e.,* `struct dl_rq`). Therefore, it is possible to extend `SCHED_DEADLINE` while keeping other scheduling policies untouched.

6.2 Implementation of ϵ-Scheduler

We implement the ϵ-Scheduler as a scheduling mode under the existing `SCHED_DEADLINE`. The mode can be switched by setting a custom kernel parameter `/proc/sys/kernel/sched_dl_mode`. The ϵ-Scheduler's main functionality is implemented in the function `replenish_dl_entity()` that is invoked whenever a new job of a real-time task arrives. In this function, the ϵ-Scheduler generates a randomized inter-arrival time based on the Laplace distribution associated with the current task (described below). The generated inter-arrival time is used to compute the deadline for the newly arrived job. This value is also used in the function `start_dl_timer()` to schedule the arrival of the next job.

Laplace Distribution. ϵ-Scheduler requires the generation of random numbers based on Laplace distribution for obtaining randomized inter-arrival times. However, the Linux kernel code is self-contained (*i.e.,* it does not depend on the standard or any other C libraries) and thus a random number generator that's based on Laplace distribution is not natively supported. While it is possible to build such a generator out of the existing random number generation function `get_random_bytes()`, the required computations(*e.g.,*

[4] https://github.com/epsilon-scheduler
[5] https://www.raspberrypi.org/products/raspberry-pi-4-model-b/.
[6] https://wiki.linuxfoundation.org/realtime/
[7] This change in system variable settings was mainly configured for the purpose of experimenting with the ideas of ϵ-Scheduler only. For most real use-cases, users can keep this system variable untouched for more flexibility.



(a) ϵ-Scheduler ($\epsilon = 10^3$) (b) ϵ-Scheduler ($\epsilon = 10$)

Figure 5: The trajectory of the autonomous rover through predefined way points running under ϵ-Scheduler. The result indicates that larger diversification and higher protection with $\epsilon = 10$ can result in larger offsets in trajectory. The worst observed deviations are $0.027m$ and $0.057m$ in the cases of $\epsilon = 10^3$ and $\epsilon = 10$ respectively, compared to the trajectory of Vanilla EDF. These deviations are reasonably small and the autopilot performance is deemed acceptable.

logarithm calculations) will be costly. Considering that the task set parameters are fixed at the design stage, the Laplace distributions needed by each task are fixed and known as well. Therefore, rather than building a common Laplace distribution-based random number generator, we may convert each required Laplace distribution's percent point function (PPF) into an array and store each of them in the kernel. Then, a Laplace distribution-based random number can be drawn by randomly picking (with using `get_random_bytes()`) a number from the array that's associated with the desired Laplace distribution. The details of the aforementioned conversion and the algorithm for the PPF-based Laplace distribution random number generator are presented in Appendix B.

While this method allows us to draw a Laplace distribution-based random number with a cost of a `get_random_bytes()` call, each distribution requires some memory to store an array converted from the PPF. Yet, as demonstrated by our implementation, an `u32` (*i.e.,* `unsigned int`) array storing 100 PPF points (which takes up to 400 bytes) is sufficient to produce the desired distribution. An example of the histogram for the generated random inter-arrival times drawn by the implemented ϵ-Scheduler in RT Linux for a task with a target period $100ms$ can be found in Figure 11 in Appendix.

7 EVALUATION ON REAL APPLICATIONS

In this section we evaluate the ϵ-Scheduler with using two diverse, real applications to demonstrate its usability and understand its security and performance impact in a real-world setting. A design space exploration using simulated tasks is presented in Section 8.

7.1 Autonomous Rover System

7.1.1 Experiment Setup. We first conducted a set of experiments on a 1/24 scale rover running an autopilot application, RoverBot[8], on the RPi4 platform introduced in Section 6. The autopilot application consists of 7 tasks (*i.e.,* Actuator, RCInput, BatteryMonitor, AHRS, Localizer, Navigator and GroundControl). Each task runs as a process in Linux and can be configured as a real-time or non-real-time task. The system is equipped with an Intel RealSense T265 tracking camera[9] that enables precise indoor localization as well as indoor navigation. With such features, we let the rover steer through a series of predefined way points that form a closed loop and record the resulting trajectory under both *(a)* Vanilla EDF and *(b)* ϵ-Scheduler, both with $\epsilon = 10$ and $\epsilon = 10^3$. An additional test that uses a different set of predefined way points and associated results are presented in Appendix C.1 for reference. In each test, we let the rover run three rounds following the ways points. To analyze the performance of the system, we focus on adding noise to the Actuator task that receives control commands and sends PWM updates for driving, steering and throttle (at $100Hz$), while keeping other tasks as non-real-time tasks.

Table 1: K-S Test and Average Minimum L2 Distance

Way Points	Comparison	K-S	p-val	Min Dist (Meters)
Irregular	$\epsilon = 10^3$, vanilla	0.016512	0.9997	0.006885
	$\epsilon = 10$, vanilla	0.015882	0.9998	0.009288
	$\epsilon = 10^3$, $\epsilon = 10$	0.018564	0.9982	N/A

7.1.2 Results. The experiment results are shown in Figure 5. In all test cases, the rover always starts at the coordinate $(0, 0)$. As the results suggest, $\epsilon = 10$ demonstrated a larger deviation in the trajectory compared to $\epsilon = 10^3$. The mean task frequency is $65.06Hz$ with $\epsilon = 10^3$ and $10.22Hz$ with $\epsilon = 10$. On the other hand, the trajectories show that the rover is still able to hit the target way points in both $\epsilon = 10^3$ and $\epsilon = 10$ cases. In particular, the trajectory of $\epsilon = 10^3$ matches that of Vanilla EDF with small deviations. This shows that the ϵ-Scheduler can be applied to real applications and also meet users' needs (*e.g.,* better protection or better performance) using the adjustable ϵ parameter.

Table 1 shows the values obtained from the kolmogorov-smirnov (K-S) tests [45] (detailed in Appendix C.2) and the average minimum distance between the paths followed by the rover with the respective schedulers. The very small K-S statistic values and very large corresponding p-values confirm that the rover paths with both ϵ-Schedulers ($\epsilon = 10^3$ and $\epsilon = 10$), closely follow the original way points as Vanilla EDF and that the former two paths closely resemble each other. The last column in the table shows the average minimum distance between a point in the observed path (with ϵ-Scheduler $\epsilon = 10^3$ or $\epsilon = 10$) and a point in the reference path (Vanilla EDF). Firstly, for each point in the observed path, we find a point in the reference path that corresponds to the minimum distance. Then, we simply take the average of all such minimum distances. The very small values of average minimum distances show that the observed paths closely follow the reference path.

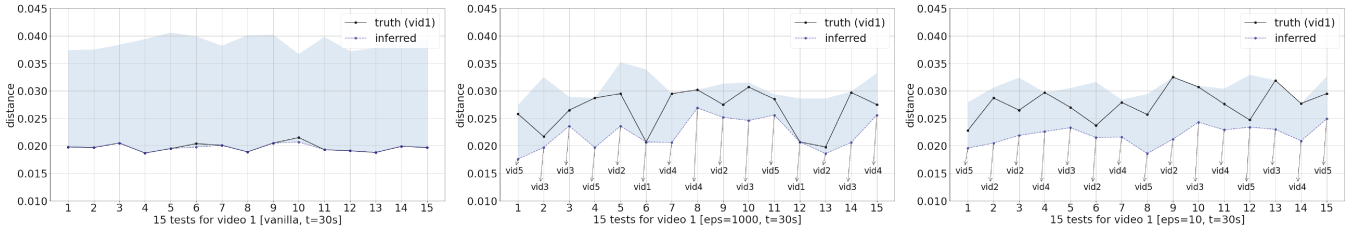
Hence, we can conclude from our evaluation above that although running the rover with our ϵ-Scheduler ($\epsilon = 10^3$ or $\epsilon = 10$) causes small deviations from the expected trajectory, the deviations themselves are negligible, making the performance drop relatively insignificant. However, the security improves greatly due to the randomization introduced by our ϵ-Scheduler.

7.2 Video Streaming over the Internet

7.2.1 Experiment Setup. We conducted another set of experiments to test the effectiveness of our ϵ-Scheduler on RTS. We built a video streaming application using Dynamic Adaptive Streaming

[8]https://github.com/bo-rc/Rover

[9]https://www.intelrealsense.com/tracking-camera-t265



(a) Vanilla EDF **(b) ϵ-Scheduler ($\epsilon = 10^3$)** **(c) ϵ-Scheduler ($\epsilon = 10$)**

Figure 6: The similarity distance measures for 15 traffic samples with the server running on RPi4. We shaded the region between the maximum and the minimum distances. Clearly, the similarity distances between the traffic patterns and video 1's fingerprints are rarely the minimum across the 15 tests with ϵ-Scheduler (2 out of 15 with $\epsilon = 10^3$ and 0 out of 15 with $\epsilon = 10$), while they are the minimum with Vanilla EDF in 13 out of 15 tests. This shows that the random noise added to the traffic patterns under ϵ-Scheduler reduces the effectiveness of the traffic-based video identification attack.

Table 2: FPS Observed by The Client (Video 1)

Scheduler	Max	Mean	Min	Std	CV
Vanilla EDF	32.00	29.90	28.00	0.68	0.02
ϵ-Scheduler ($\epsilon = 10^3$)	28.00	25.87	22.00	1.71	0.07
ϵ-Scheduler ($\epsilon = 10$)	14.00	9.23	6.00	2.07	0.22

over HTTP (DASH) as the video streaming standard and flask [24] for our web application. Our goal was to show that the ϵ-Scheduler is useful in negating traffic-based attacks (types of data leakage attacks as presented by Gu *et al.* [25][10]) on such video streaming applications without significantly affecting the performance of the application itself. The video stream is hosted by a server and the client is the receiver of the video stream that is transmitted via the application over the internet over a distance of 1800 miles. Our attacker is placed in between the server and the client, so that eavesdropping on the network traffic can be easily carried out. The performance of the application is measured using the frames per second (FPS) of the video. Ideally, the FPS of the video at the client's end is similar to the FPS of the video sent from the server. It is important to note that the ϵ-Scheduler only randomizes the arrival time of video frames to the client and does not change the content of the video. An attack can be devised on such applications by exploiting some key properties of DASH video streaming. The details of how the attack works are presented in Appendix D.

In this experiment, we set ϵ-Scheduler with $\Delta\eta = 190ms$, $J = 16$ with the desired protection duration to be $\lambda = 500ms$[11] for the video streaming task running at $30Hz$. Our evaluation verifies whether the video identification attack in the case of the ϵ-Scheduler shows results that are random at best. In our setup, we use five videos with varying content, frame rates and resolutions (see Appendix D). We consider a total of five streaming scenarios, each scenario being the event when only one of the five videos is being streamed via our application, *i.e.,* only video x is being streamed, where $x \in \{1, 2, ..., 5\}$. Eavesdropping is done for 30 seconds with 2 seconds as the segment length and the corresponding traffic patterns are captured using Wireshark. For repeatability, this is done fifteen times for each scheduler (Vanilla EDF, ϵ-Scheduler with $\epsilon = 10$ and $\epsilon = 10^3$), resulting in 45 traffic pattern samples for each scenario.

[10]We created the attack from scratch as the authors denied us access to the source code.

[11]It is shown that the security tasks are typically assigned periods in the range $[250ms, 500ms]$ [27]. In our evaluation, we take $500ms$ (*i.e.,* the worst protection) to estimate protection duration J.

7.2.2 Experiment Results. For identification purposes, we calculate the distance metric *dist*, which is representative of the similarity between a given traffic pattern and a video fingerprint. Hence, given a traffic pattern and a dataset containing n videos, there are n distances generated ($n = 5$ in our case). The smaller the value of *dist*, the greater the probability for the traffic pattern matching the video fingerprint, which is equivalent to the probability of the corresponding video being streamed during the eavesdropping. In order to compensate for our relatively short eavesdropping time, instead of setting thresholds on distances to identify the target video as done by Gu *et al.* [25], we simply identify the target video as being the one that had the minimum distance out of the five calculated distances. Figure 6 shows the similarity distances for the traffic samples collected when video 1 was being streamed. The results obtained when the other videos (2,3,4 and 5) were being streamed, closely match that of video 1.

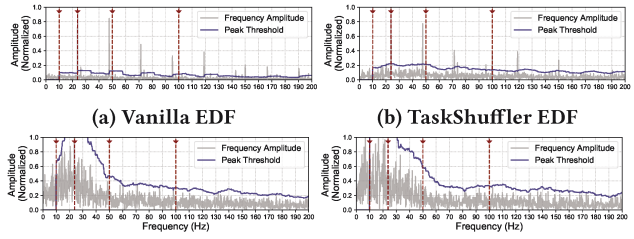
Table 2 shows the FPS statistics observed at the client's end over a duration of 30 seconds for video 1 in which an FPS data point is computed using the number of frame packets received from the Internet per 0.5 seconds. The FPS of video 1 sent from the server is 30. The CV (Coefficient of Variation) value represents relative variability of the FPS in each scheduler configuration. It reveals that FPS in the case of Vanilla EDF has the smallest variability as there is no randomization while it shows the largest variability in the case of ϵ-Scheduler with $\epsilon = 10$. In the case of ϵ-Scheduler with $\epsilon = 10^3$, it has a reasonably small CV value and slightly decreased mean FPS with a good protection against the eavesdropping attack that's comparable to $\epsilon = 10$ (see Figure 6). As a result, it shows that a balance between performance (*i.e.,* FPS) and security can be reached by using ϵ-Scheduler with $\epsilon = 10^3$ in this experiment.

8 DESIGN SPACE EXPLORATION

Besides the evaluation with real applications, we also conduct an evaluation with using simulations as well as a real hardware platform (*i.e.,* RPi4). The simulation enables us to explore a larger design space while the hardware platform enables us to understand the true scheduling overheads in a realistic environment.

8.1 Experiment Setup

8.1.1 Simulation. A set of simulated tasks with timing parameters of avionics system [49] (total task utilization 0.64) is used to



(a) Vanilla EDF **(b) TaskShuffler EDF**

(c) ϵ-Scheduler ($\epsilon = 10^3$) **(d) ϵ-Scheduler ($\epsilon = 10$)**

Figure 7: Results of the DFT analysis for the avionics tasks (Table 6 in Appendix). The blue lines are the normalized amplitudes for the corresponding frequency bins and the green lines are the Z-scored based moving peak threshold for detecting outstanding peaks. The results suggest that ϵ-Scheduler creates a wide range of noise in the frequency spectrum and is effective in obscuring the periodic elements enclosed in the original schedule.

examine the outcome of the ϵ-Scheduler in the first part of our evaluation. The tasks' parameters are shown in Table 6 in Appendix. The ϵ-Scheduler is also tested extensively using simulation tasks generated from a mechanism is commonly used in literature [7, 14, 27, 60]. The details for the generation of the 6000 tested task sets are provided in Appendix E.

To explore the best-case protection as well as the impact on the system performance, we configure the extended task parameters to achieve the task-level indistinguishability. The efficacy of job-level indistinguishability is specifically examined against the ScheduLeak attack [14] (results presented in Section 8.2.3). To achieve the task-level indistinguishability, $\Delta\eta$ is assigned to $200ms - 10ms = 190ms$. J_i for each task is calculated using Equation 8 with a protection duration of $500ms$ (demonstrated to be practical to perform periodic security checks RTS [27]). We consider two ϵ settings 10 and 10^3 that represent values that one may reasonably choose based on the noise range shown in Figure 4. In our experiments, we use a fixed simulation duration ($5000ms$) so that we are able to compare the experiment results across different task sets.

We also include the vanilla EDF scheduler and a state-of-the-art randomization-based scheduler for comparison. The randomization-based scheduler, labeled as "TaskShuffler EDF", is an EDF-based scheduler that ports the TaskShuffler's randomization protocol.

8.1.2 Measuring Scheduling Overheads. To evaluate the scheduling overheads, we conduct experiments on the RPi4 platform running RT Linux. We use the built-in `SCHED_DEADLINE` scheduler as the Vanilla EDF scheduler and an implementation of TaskShuffler EDF for comparison. The timing overheads for a function is measured using the `trace-cmd` command. For evaluating power consumption, we use a High Voltage (HV) Power Monitor manufactured by Monsoon[12] that supplies a $5.2V$ power to the RPi4 board. The power consumption is then monitored in the monitor's software, PowerTool version 5.0.0.25.

8.2 Experiment Results

8.2.1 Discrete Fourier Transform Analysis. First we try to understand the (deterministic) periodicity in the schedules produced by: *(a)* Vanilla EDF scheduler, *(b)* TaskShuffler EDF scheduler and *(c)* our ϵ-Scheduler (with $\epsilon = 10^3$ and $\epsilon = 10$). Since we are concerned about the periodic components in the task schedules, frequency spectrum analysis tools such as Discrete Fourier Transform (DFT) [42] can be useful (the details of measurement for DFT are provided in Appendix F).

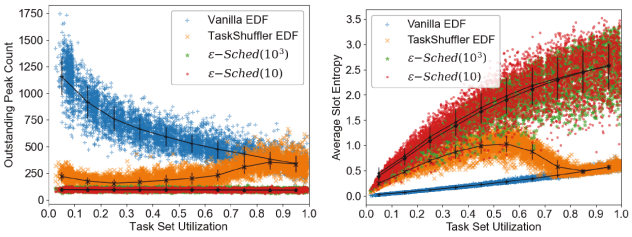
We conduct the DFT analysis on the schedules based on the avionics task set (Table 6 in Appendix) and the resulting frequency spectra are shown in Figure 7. As revealed by the peaks displayed in Figure 7(a), the task periods are easily identifiable in the schedule generated by the vanilla EDF scheduler because of its work-conserving nature. It's worth pointing out that the $100ms$ (*i.e.,* $10Hz$) value does not show up as a peak in the spectrum because the corresponding task has a very small execution time (*i.e.,* $0.002ms$). Figure 7(b) shows the spectrum of the same task set scheduled under the TaskShuffler EDF scheduler and the result is similar to the vanilla EDF scheduler except with more base noise. This is due to the high task set utilization (*i.e.,* 0.64 in this case) that provides fewer opportunities for obfuscating the schedule. While the task set may not be exhaustive, it does demonstrate the shortcoming of the TaskShuffler's randomization protocol – it gets less effective when the system utilization is high. This shortcoming can also be seen in later experiments. On the other hand, Figure 7(c) and (d) show the spectra when scheduled using the ϵ-Scheduler with $\epsilon = 10^3$ and $\epsilon = 10$, respectively. Both settings add significant noise across the entire frequency domain. As a result, no peaks stand out, especially ones that match the task frequencies.

The green lines shown in Figure 7 are the moving peak threshold calculated using the Z-score based peak detection algorithm (see Appendix F for details). From the figures we can see that the threshold is useful for identifying the outstanding peaks while filtering out background noise. The outstanding peaks represent the true periodicity coming out of the schedule and thus are particularly useful for attackers to reconstruct timing information. Intuitively, the more outstanding peaks that are collected, the more precise information the attackers have available to them.

Next we use the aforementioned peak detection algorithm to count the number of outstanding peaks and test with extensive simulations to get a broader understanding of the effectiveness of ϵ-Scheduler in obscuring the task periodicity. The experiment results are presented in Figure 8(a) where each point represents the result of a task set for the corresponding scheduler. As expected, the vanilla EDF scheduler yields systems with stronger periodicity and more peaks that stand out. On the other hand, the TaskShuffler EDF scheduler can effectively obscure the task periodicity for most of the task sets *except those with higher utilization.* With ϵ-Scheduler, no significant peaks are detected due to the addition of larger overall noise in both $\epsilon = 10^3$ and $\epsilon = 10$ settings. The result also demonstrates that the efficacy of ϵ-Scheduler is independent of the task utilization, in contrast with Vanilla and TaskShuffler.

8.2.2 Average Slot Entropy. Next we analyze the schedules by measuring their average slot entropy. The notion of *Schedule Entropy* was first introduced to calculate the randomness given to a task schedule by the TaskShuffler scheduling algorithm [60]. They then proposed the *Upper-Approximated Schedule Entropy*, $\widetilde{H}_\Gamma(\mathcal{S})$, to empirically estimate the schedule entropy of a given task set. A

[12] https://www.msoon.com/high-voltage-power-monitor



(a) Outstanding peak count (b) Average slot entropy

Figure 8: The results indicate (a) Vanilla EDF yields a large number of peaks that are useful for adversaries to learn the schedule while there are no significant amount of peaks detected with ϵ-Scheduler and (b) ϵ-Scheduler generates diversified schedules with higher entropy (*i.e.,* more randomness).

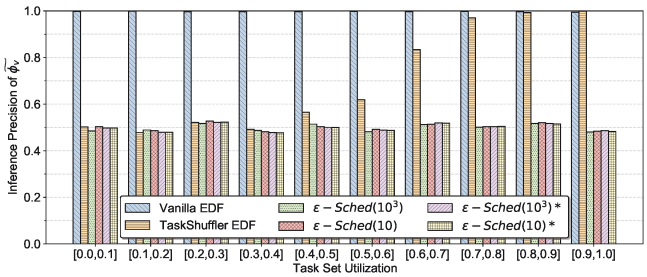


Figure 9: The inference precision results of $10 \cdot LCM(T_o, T_v)$ grouped by the task set utilization. The experiment suggests that ϵ-Scheduler can offer effective protection against the ScheduLeak attack. Such an effect is independent to the attack duration and the task set utilization.

bound is then derived by Vreman *et al.* [56] showing the legitimacy of such estimation. As the scale of the entropy depends on the length of the schedule under analysis, in this paper we use *Average Slot Entropy* [56] that calculates the mean slot entropy based on the upper-approximated schedule entropy.

The results are shown in Figure 8(b). Similar to Figure 8(a), a point represents the average slot entropy of a task set under the corresponding scheduler. The results indicate that ϵ-Scheduler *yields higher entropy* than the other two schedulers even when the system utilization is high (note that TaskShuffler EDF fails to obfuscate the schedules in these instances). For the ϵ-Scheduler, $\epsilon = 10$ generally performs better than $\epsilon = 10^3$ *w.r.t.* the entropy as the former has a wider variation range for the noise-enhanced inter-arrival times.

8.2.3 Inference Precision. To understand the effectiveness of our mechanisms against scheduler side-channel attacks, we carry out the ScheduLeak attacks [14] against the simulation tasks. The metric, *inference precision* [14, Definition 2], denoted by $\mathbb{I}_v^o$, was introduced to evaluate the effectiveness of a side-channel attack *w.r.t.* the task phase inference. It represents the precision of the inferred phase $\widetilde{\phi_v}$ compared to the true phase of a target task ϕ_v. The inference precision is calculated by $\mathbb{I}_v^o = \left| \frac{\Delta\phi}{\frac{T_v}{2}} - 1 \right|$ A larger $\mathbb{I}_v^o$ indicates that the inference $\widetilde{\phi_v}$ is more precise in inferring ϕ_v.

To illustrate, let us consider a task set consisting of N tasks $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$ whose task IDs are ordered by their periods (*i.e.,* $T_1 > T_2 > ... > T_n$). The observer (attacker) task is then selected as

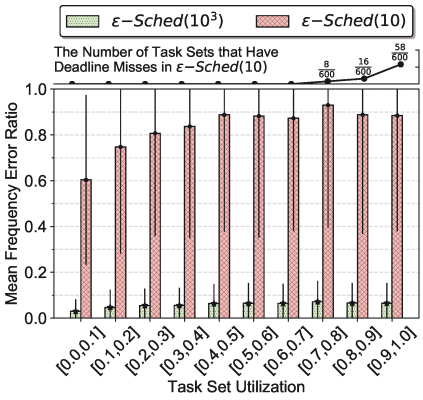


Figure 10: Results of the mean frequency error ratio grouped by the system utilization. It shows that a large ϵ value can lead to greater mean frequency error and also cause some tasks to miss deadlines when the utilization is high, as displayed by the plot at the top section that shows the number of task sets that have experienced deadline misses in each utilization group with $\epsilon = 10$.

the $(\lfloor \frac{n}{3} \rfloor + 1)$-th task and the victim task is selected as the $(n - \lfloor \frac{n}{3} \rfloor)$-th task. This assignment ensures that there exist other tasks with diverse periods (*i.e.,* some with smaller periods and some with larger periods compared to T_o and T_v.)

We first run experiments for achieving task indistinguishability. The results suggest that ScheduLeak shows better inference precision as attack duration increases for both vanilla EDF and TaskShuffler. On the other hand, ϵ-Scheduler *offers consistent protection throughout the entire course of the attack* (a plot for this experiment result is provided in Appendix Figure 15). Figure 9 shows the breakdown of the inference precision results grouped by the utilization. It reveals that the TaskShuffler scheduler offers less effective protection when the utilization is high due to limited possible randomization. On the other hand, our ϵ-Scheduler yields consistent performance across all task utilizations leading to an average inference precision (0.498 and 0.501 for $\epsilon = 10^3$ and $\epsilon = 10$ respectively) that is *close to the outcome produced by a random guess.*

Next, we test if job indistinguishability for the victim task is sufficient to protect it against ScheduLeak. Here, all tasks have consistent inter-arrival times based on their periods (*i.e.,* $\epsilon_i = \infty$) *except* the victim task. The results are presented as the 5^{th} (ϵ-*Sched*$(10^3)^*$) and 6^{th} (ϵ-*Sched*$(10)^*$) bars in each group shown in Figures 9 and 15. As shown, the victim task is protected by job indistinguishability. The ScheduLeak attack fails to take advantage of the side-channels and yields inference precision at a level similar to a random guess.

8.2.4 QoS-Based Results. While the above results show that the ϵ-Scheduler is effective in increasing the noise in the schedule, we are interested in the impact on the QoS of the tasks. We first examine the case of deadline misses in our experiments. As expected, both Vanilla EDF and TaskShuffler EDF obey strict real-time constraints and thus do not experience any deadline misses. In ϵ-Scheduler, no deadline miss has been observed when $\epsilon = 10^3$. However, in the case of $\epsilon = 10$, we observe intermittent deadline misses in some of task sets with high utilization. The number of task sets that have encountered deadline misses in such a setting is plotted at the top

Table 3: Summary of Scheduling Overhead Measurement

Mean of Measurement	Vanilla EDF	T.S. EDF	$\epsilon = 10^3$	$\epsilon = 10$
Context Switch Count Ratio	1	2.525	0.914	0.696
`pick_next_dl_entity()`	$1.25us$	$4.3us$	$1.44us$	$1.39us$
ϵ-Scheduler function*	-	-	$5.79us$	$5.41us$
Power Usage (`performance`)	$2.37W$	$2.39W$	$2.38W$	$2.36W$
Power Usage (`ondemand`)	$2.20W$	$2.3W$	$2.08W$	$2.05W$

* `get_next_inter_arrival_time()`

of Figure 10. As the result shows, only 1.37% of the tested task sets have deadline misses. Among these cases, no consecutive deadline miss has been observed.

We next examine how close to the indistinguishable tasks perform to the desired frequencies. A task's frequency error is calculated as the difference between the task's mean and desired frequencies. The mean of the frequency errors (grouped by task set utilization) is shown in the bar chart in Figure 10. The result indicates that task sets scheduled by ϵ-Scheduler with $\epsilon = 10$ has frequency error significantly larger than that with $\epsilon = 10^3$. It is expected as $\epsilon = 10$ yields a wider inter-arrival range (more noise added). It is also worth pointing out that the frequency error is due to the bounds in generating the noise-added inter-arrival times that can lead to an asymmetric distribution (as an example, see the distribution for $\mu = 33.3ms$ in Figure 4(b)).

While the mean task frequency gives us an insight into the overall performance of the service delivery, it is crucial to know how often the task is performing at a frequency below what is expected (*i.e.,* with inter-arrival times larger than the desired period) as such execution usually has a direct impact on the task's commitment to the service delivery. We measure such a property for a task by calculating the ratio of the number of under-performing inter-arrival times to the total number of generated inter-arrival times. In this experiment, we first compute the worst under-performance ratio for each task set (by measuring it for each task and selecting the worst in the task set) and then calculate the mean of the worst under-performance ratios. From our experiment results, the under-performance ratio can be biased towards 0.5 and above, leading to a degradation in the execution frequency (a detailed experiment plot is provided in Appendix, Figure 16). This usually happens to the task that has a small target period and hence, an asymmetric distribution that tends to generate larger inter-arrival times (again, see Figure 4(b) for an example). It hints that one should expect a degradation in the service when using ϵ-Scheduler, particularly with a small ϵ value (*i.e.,* larger noise and variation in the schedule).

8.2.5 Scheduling Overhead. We next evaluate the scheduling overhead of the ϵ-Scheduler, together with the Vanilla EDF and TaskShuffler EDF schedulers as a comparison. The measurement results are summarized in Table 3.

We set Vanilla EDF as the base and calculate the context switch count ratio compared to TaskShuffler EDF and ϵ-Scheduler in simulation. The result suggests that TaskShuffler EDF generates a twofold increase in the number of context switches. This matches the design of the TaskShuffler's randomization protocol that aims to obfuscate the schedule by introducing more scheduling points (*i.e.,* more context switches). On the other hand, ϵ-Scheduler *produces fewer context switches* as the generated inter-arrival times can be greater than Vanilla EDF (*i.e.,* task executing less frequently). This measurement generally matches the result shown in Figure 16.

Next, we execute a set of tasks (with parameters given in Table 6) on RT Linux on the RPi4 platform to measure the mean cost of scheduling. We first measure the execution time overheads of the main scheduling function `pick_next_dl_entity()` that picks the next task at a scheduling point. The result shows that TaskShuffler EDF has larger overhead as it invokes `get_random_bytes()` that takes an average $2.23us$ to generate a 64-bit random number. On the other hand, the ϵ-Scheduler has overheads that are very similar to Vanilla EDF as the scheduling mechanism is identical in `pick_next_dl_entity()`. To evaluate the true overhead of ϵ-Scheduler, we measure `get_next_inter_arrival_time()` where randomized inter-arrival times are generated in our ϵ-Scheduler implementation. As shown in the table, the time cost is around $5.79us$ and is independent of the ϵ setting. This cost is mainly due to the invocation of the random number generation function, `get_random_bytes()`. Note that this overhead is incurred in the scheduler when a job arrives, which is not equivalent to the context switch overhead as an arrival of a job in EDF (and ϵ-Scheduler) does not necessarily lead to a new scheduling event (*i.e.,* a `pick_next_dl_entity()` call).

We also measured the power consumption of the platform for each of the schedulers. When the scaling governor is configured as `scaling_governor = performance` (a typical setting for RTS to maintain a predictable execution time and behavior), the power consumption consistent for all schedulers. It is expected as the CPU runs at the highest frequency at all times under the `performance` setting. For a comparison purposes, we measure the power consumption with `scaling_governor = ondemand` that lowers the CPU frequency (*i.e.,* less power consumption) when idling for a significant amount of time. The resulting power consumption matches what we have learned from the above experiments (*e.g.,* lower context switch ratio in ϵ-Scheduler) and suggest that ϵ-Scheduler does not result in higher power consumption.

9 DISCUSSION AND CONCLUSION

From the evaluation, both $\epsilon = 10^3$ and $\epsilon = 10$ settings produce promising results for obscuring the periodicity and diversifying the schedule. However, as shown by the QoS measurements, the $\epsilon = 10^3$ setting yields more reasonable variations in the task frequencies. While ϵ-Scheduler offers less protection with $\epsilon = 10^3$ value, it may not be unusual to choose such a large ϵ value in many RTS. The same outcomes can be drawn from the evaluation using the two real applications presented in Section 7. In both applications, the system reaches a balance between performance and security with $\epsilon = 10^3$. On the other hand, with $\epsilon = 10$, the results demonstrate how diverse the performance impact could be for different applications. In such a setting, the rover system performs with an acceptable error, while the video streaming service becomes unusable. Considering that every application has its unique tolerance to variations, the ϵ value *should be determined on a case-by-case basis in conjunction with system designers.*

A possible improvement is to dynamically adjust the ϵ value based on the QoS and protection demand at run-time. In such a case, ϵ is particularly useful as a security parameter to be integrated with a feedback control real-time scheduling algorithm (*e.g.,* [43]).

REFERENCES

[1] Fardin Abdi, Chien-Ying Chen, Monowar Hasan, Songran Liu, Sibin Mohan, and Marco Caccamo. 2018. Guaranteed physical security with restart-based design for cyber-physical systems. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. IEEE Press, 10–21.

[2] Fardin Abdi, Chien-Ying Chen, Monowar Hasan, Songran Liu, Sibin Mohan, and Marco Caccamo. 2018. Preserving Physical Safety Under Cyber Attacks. *IEEE Internet of Things Journal* (2018).

[3] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. 2002. The EM side—channel (s). In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 29–45.

[4] Hyeongboo Baek and Chang Mook Kang. 2020. Scheduling Randomization Protocol to Improve Schedule Entropy for Multiprocessor Real-Time Systems. *Symmetry* 12, 5 (2020), 753.

[5] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. 2006. The sorcerer's apprentice guide to fault attacks. *Proc. IEEE* 94, 2 (2006), 370–382.

[6] Donald J Berndt and James Clifford. 1994. Using dynamic time warping to find patterns in time series.. In *KDD workshop*, Vol. 10. Seattle, WA, USA:, 359–370.

[7] Marko Bertogna and Sanjoy Baruah. 2010. Limited preemption EDF scheduling of sporadic task systems. *IEEE Trans. on Ind. Info.* 6, 4 (2010), 579–591.

[8] Enrico Bini and Giorgio C Buttazzo. 2005. Measuring the performance of schedulability tests. *RTS Journal* 30, 1-2 (2005), 129–154.

[9] Alan Burns and Stewart Edgar. 2000. Predicting computation time for advanced processor architectures. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*. IEEE, 89–96.

[10] Nathan Burow, Ryan Burrow, Roger Khazan, Howard Shrobe, and Bryan C Ward. 2020. Moving Target Defense Considerations in Real-Time Safety-and Mission-Critical Systems. In *Proceedings of the 7th ACM Workshop on Moving Target Defense*. 81–89.

[11] Defense Use Case. 2016. Analysis of the cyber attack on the Ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)* (2016).

[12] Francisco J Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, et al. 2013. Proartis: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems* (2013).

[13] Konstantinos Chatzikokolakis, Miguel E Andrés, Nicolás Emilio Bordenabe, and Catuscia Palamidessi. 2013. Broadening the scope of differential privacy using metrics. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 82–102.

[14] 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.

[15] Thomas M. Chen and Saeed Abu-Nimeh. 2011. Lessons from Stuxnet. *Computer* 44, 4 (April 2011), 91–93.

[16] Hoon Sung Chwa, Kang G Shin, and Jinkyu Lee. 2018. Closing the gap between stability and schedulability: a new task model for Cyber-Physical Systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 327–337.

[17] Jorge Cortés, Geir E Dullerud, Shuo Han, Jerome Le Ny, Sayan Mitra, and George J Pappas. 2016. Differential privacy in control and network systems. In *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE, 4252–4272.

[18] Cynthia Dwork. 2008. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*. Springer, 1–19.

[19] Cynthia Dwork, Aaron Roth, et al. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407.

[20] Iker Esnaola-Gonzalez, Meritxell Gómez-Omella, Susana Ferreiro, Izaskun Fernandez, Ignacio Lázaro, and Elena García. 2020. An IoT Platform towards the Enhancement of Poultry Production Chains. *Sensors* 20, 6 (2020), 1549.

[21] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. 2009. An EDF scheduling class for the Linux kernel. In *Real-Time Linux Wkshp*.

[22] A. Ghassami, X. Gong, and N. Kiyavash. 2015. Capacity limit of queueing timing channel in shared FCFS schedulers. In *2015 IEEE International Symposium on Information Theory (ISIT)*. 789–793. https://doi.org/10.1109/ISIT.2015.7282563

[23] Xun Gong and Negar Kiyavash. 2016. Quantifying the Information Leakage in Timing Side Channels in Deterministic Work-conserving Schedulers. *IEEE/ACM Trans. Netw.* 24, 3 (June 2016), 1841–1852. https://doi.org/10.1109/TNET.2015.2438860

[24] Miguel Grinberg. 2018. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.".

[25] Jiaxi Gu, Jiliang Wang, Zhiwen Yu, and Kele Shen. 2019. Traffic-Based Side-Channel Attack in Video Streaming. *IEEE/ACM Transactions on Networking* 27, 3 (2019), 972–985.

[26] Jeffery Hansen, Scott A Hissam, and Gabriel A Moreno. 2009. Statistical-based wcet estimation and validation. In *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*.

[27] Monowar Hasan, Sibin Mohan, Rakesh B Bobba, and Rodolfo Pellizzoni. 2016. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 123–134.

[28] Jianping He and Lin Cai. 2016. Differential private noise adding mechanism and its application on consensus. *arXiv preprint arXiv:1611.08936* (2016).

[29] Naoise Holohan, Spiros Antonatos, Stefano Braghin, and Pól Mac Aonghusa. 2018. The bounded Laplace mechanism in differential privacy. *arXiv preprint arXiv:1808.10410* (2018).

[30] Zhenqi Huang, Yu Wang, Sayan Mitra, and Geir E Dullerud. 2014. On the cost of differential privacy in distributed control systems. In *Proceedings of the 3rd international conference on High confidence networked systems*. 105–114.

[31] Aini Hussain, M. A. Hannan, Azah Mohamed, Hilmi Sanusi, and A. K. Ariffin. 2006. Vehicle crash analysis for airbag deployment decision. *International Journal of Automotive Technology* 7, 2 (2006), 179–185.

[32] Damir Isovic. 2001. *Handling Sporadic Tasks in Real-time Systems: Combined Offline and Online Approach*. Mälardalen University.

[33] Ke Jiang, L. Batina, P. Eles, and Zebo Peng. 2014. Robustness Analysis of Real-Time Scheduling Against Differential Power Analysis Attacks. In *2014 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 450–455. https://doi.org/10.1109/ISVLSI.2014.11

[34] S. Kadloor, N. Kiyavash, and P. Venkitasubramaniam. 2016. Mitigating Timing Side Channel in Shared Schedulers. *IEEE/ACM Transactions on Networking* 24, 3 (June 2016), 1562–1573. https://doi.org/10.1109/TNET.2015.2418194

[35] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. 2010. Experimental Security Analysis of a Modern Automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*. 447 –462. https://doi.org/10.1109/SP.2010.34

[36] Kristin Krüger, Marcus Völp, and Gerhard Fohler. 2018. Vulnerability Analysis and Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS)*. 22:1–22:17. https://doi.org/10.4230/LIPIcs.ECRTS.2018.22

[37] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. 2008. Predictable Programming on a Precision Timed Architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 137–146.

[38] Bruno Monteiro Rocha Lima, Luiz Claudio Sampaio Ramos, Thiago Eustaquio Alves de Oliveira, Vinicius Prado da Fonseca, and Emil M Petriu. 2019. Heart Rate Detection Using a Multimodal Tactile Sensor and a Z-score Based Peak Detection Algorithm. *CMBES Proceedings* 42 (2019).

[39] C. L. Liu and J. W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM* (1973).

[40] Fang Liu. 2016. Statistical Properties of Sanitized Results from Differentially Private Laplace Mechanism with Univariate Bounding Constraints. *arXiv preprint arXiv:1607.08554* (2016).

[41] Fang Liu. 2018. Generalized gaussian mechanism for differential privacy. *IEEE Transactions on Knowledge and Data Engineering* 31, 4 (2018), 747–756.

[42] Songran Liu, Nan Guan, Dong Ji, Weichen Liu, Xue Liu, and Wang Yi. 2019. Leaking your engine speed by spectrum analysis of real-Time scheduling sequences. *Journal of Systems Architecture* (2019).

[43] Chenyang Lu, John A Stankovic, Sang H Son, and Gang Tao. 2002. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems* 23, 1-2 (2002), 85–126.

[44] Pau Marti, Caixue Lin, Scott A Brandt, Manel Velasco, and Josep M Fuertes. 2004. Optimal state feedback based resource allocation for resource-constrained control tasks. In *25th IEEE International Real-Time Systems Symposium*. IEEE, 161–172.

[45] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.

[46] Frank D McSherry. 2009. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 19–30.

[47] Byungho Min and Vijay Varadharajan. 2014. Design and Analysis of Security Attacks against Critical Smart Grid Infrastructures. *2014 19th International Conference on Engineering of Complex Computer Systems* 0 (2014), 59–68. https://doi.org/10.1109/ICECCS.2014.16

[48] 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.

[49] R. Pellizzoni, N. Paryab, M. Yoon, S. Bak, S. Mohan, and R. B. Bobba. 2015. A generalized model for preventing information leakage in hard real-time systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 271–282. https://doi.org/10.1109/RTAS.2015.7108450

[50] David Schneider. 2015. Jeep Hacking 101. *IEEE Spectrum* (Aug 2015). http://spectrum.ieee.org/cars-that-think/transportation/systems/jeep-hacking-101.

[51] D. Shepard, J. Bhatti, and T. Humphreys. 2012. Drone Hack: Spoofing Attack Demonstration on a Civilian Unmanned Aerial Vehicle. *GPS World* (August 2012).

[52] Joon Son and Alves-Foss. 2006. Covert Timing Channel Analysis of Rate Monotonic Real-Time Scheduling Algorithm in MLS Systems. In *2006 IEEE Information Assurance Workshop*. 361–368. https://doi.org/10.1109/IAW.2006.1652117

[53] Hugo Teso. 2013. Aicraft Hacking. In *Fourth Annual HITB Security Conference in Europe*.

[54] Nick Tsalis, Efstratios Vasilellis, Despina Mentzelioti, and Theodore Apostolopoulos. 2019. A Taxonomy of Side Channel Attacks on Critical Infrastructures and Relevant Systems. In *Critical Infrastructure Security and Resilience*. Springer, 283–313.

[55] Marcus Völp, Claude-Joachim Hamann, and Hermann Härtig. 2008. Avoiding Timing Channels in Fixed-priority Schedulers. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 44–55. https://doi.org/10.1145/1368310.1368320

[56] Nils Vreman, Richard Pates, Kristin Krüger, Gerhard Fohler, and Martina Maggio. 2019. Minimizing Side-Channel Attack Vulnerability via Schedule Randomization. In *2019 IEEE 58th Conference on Decision and Control (CDC)*. IEEE, 2928–2933.

[57] M. Völp, B. Engel, C. Hamann, and H. Härtig. 2013. On confidentiality-preserving real-time locking protocols. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 153–162. https://doi.org/10.1109/RTAS.2013.6531088

[58] Yu Wang, Zhenqi Huang, Sayan Mitra, and Geir E Dullerud. 2017. Differential privacy in linear distributed control systems: Entropy minimizing mechanisms and performance tradeoffs. *IEEE Transactions on Control of Network Systems* 4, 1 (2017), 118–130.

[59] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems* 7, 3, Article 36 (May 2008), 53 pages. https://doi.org/10.1145/1347375.1347389

[60] M. Yoon, S. Mohan, C. Chen, and L. 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

[61] Man-Ki Yoon, Bo Liu, Naira Hovakimyan, and Lui Sha. 2017. VirtualDrone: virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*. ACM, 143–154.

[62] 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. https://www.usenix.org/conference/usenixsecurity21/presentation/yoon

[63] Yuting Zhang and Richard West. 2006. Process-Aware Interrupt Scheduling and Accounting. In *Proc. of the 27th IEEE International Real-Time Systems Symposium*.

A PROOF OF THEOREM 4.3

Proof. Let $\mathcal{R}^J(\tau_i, j) = \{\mathcal{R}(\tau_i, k) | j \le k < j + J\}$ be a set of $\mathcal{R}(\cdot)$ invocations. By the definition of the inter-arrival time indistinguishable, it must satisfy

$$\Pr[\mathcal{R}^J(\tau, j) \in \mathcal{W}] \le e^{\epsilon} \Pr[\mathcal{R}^J(\tau', j') \in \mathcal{W}] \tag{9}$$

for all $\tau, \tau' \in \Gamma$, $j, j' \in \mathbb{N}$ and $\mathcal{W} \subseteq \text{Range}(\mathcal{R}^J)$.

Let $w = \{\omega_k | k \in [J]\}$ be an inter-arrival time sequence generated by $\mathcal{R}^J(\tau, j)$. Then

$$\Pr[\mathcal{R}^J(\tau, j) = w] = \prod_{k \in [J]} \Pr[\mathcal{R}(\tau, j + k - 1) = \omega_k]$$

where ω_k is calculated by $\eta_\tau(\cdot) + \text{Lap}(b)$ in which b is the Laplace distribution parameter. Expanding with the probability density function, the right term in the above equation can be rewritten as

$$\prod_{k \in [J]} \frac{1}{2b} \exp(-\frac{|\omega_k - \eta_\tau(j + k - 1)|}{b})$$

Table 4: Glossary of Notations

Notation	Definition
Real-Time Task Model	
Γ	a set of real-time tasks
τ_i	a real-time task in Γ
$\mathcal{T}_i$	a set of admissible periods of τ_i
$\mathcal{D}_i$	a set of implicit, relative deadlines of τ_i
C_i	the worst-case execution time of τ_i
η_i	the inter-arrival time function of τ_i
ϵ-Scheduler Extended Model	
$\Delta\eta_i$	the inter-arrival time sensitivity of τ_i
J_i	effective protection duration for τ_i
ϵ_i	indistinguishability scale of τ_i
$\widetilde{\mathcal{R}}(\cdot)$	bounded inter-arrival time Laplace randomized mechanism

Table 5: Summary of the Implementation Platform

Artifact	**Parameters**
Platform	ARM Cortex-A72 (Raspberry Pi 4)
System Configuration	1.5 GHz 64-bit processor, 4 GB RAM
Operating System	Debian Linux (Raspbian)
Kernel Version	Linux Kernel 4.19.71-rt24-v7l+
Kernel Configuration	`CONFIG_SMP` disabled
(`make defconfig`)	`CONFIG_PREEMPT_RT_FULL` enabled
Boot Commands	`maxcpus=1`
Run-time Variables	`sched_rt_runtime_us=-1`
	`scaling_governor=performance`
Base Scheduler	`SCHED_DEADLINE`

Then

$$\frac{\Pr[\mathcal{R}^J(\tau, j) = w]}{\Pr[\mathcal{R}^J(\tau', j') = w]} = \prod_{k \in [J]} \frac{\frac{1}{2b}\exp(-\frac{|\omega_k - \eta_\tau(j+k-1)|}{b})}{\frac{1}{2b}\exp(-\frac{|\omega_k - \eta_{\tau'}(j'+k-1)|}{b})}$$
$$= \prod_{k \in [J]} \exp(\frac{|\eta_\tau(j + k - 1) - \eta_\tau(j' + k - 1)|}{b})$$

The term $|\eta_\tau(j + k - 1) - \eta_\tau(j' + k - 1)|$ represents the difference between two inter-arrival times which can be replaced with $\Delta\eta$ for the worst case (*i.e.,* the largest possible difference defined in Definition 4.2). The above becomes

$$\prod_{k \in [J]} \exp(\frac{\Delta\eta}{b}) = \exp(\sum_{k \in [J]} \frac{\Delta\eta}{b})$$
$$= \exp(\frac{J\Delta\eta}{b})$$

Using Equation 9, we can derive b from

$$\exp(\frac{J\Delta\eta}{b}) \le \exp(\epsilon)$$

$$b \ge \frac{J\Delta\eta}{\epsilon} \tag{10}$$

Therefore, the Laplace distribution with the scale $b = \frac{J\Delta\eta}{\epsilon}$ preserves ϵ-indistinguishability up to J instances. ∎

Table 6: Timing Parameters of a Avionics Demonstrator [49]

Task Name	WCET (ms)	Period (ms)
Software Control Task	2	20
Mission Planner	0.002	100
Encryption	3	42
Image Encoding	18	42
Image I/O	1.46	42
Network Manager	0.03	10

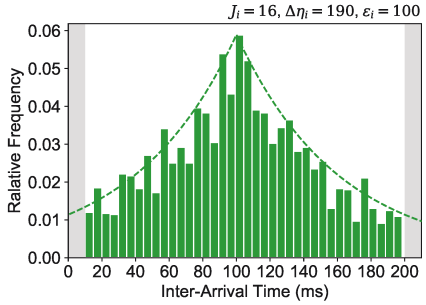


Figure 11: Histogram of the randomized inter-arrival times generated by ϵ-Scheduler for the task τ_i with a desired period $100ms$ running in RT Linux. The extended task parameters are assigned to be $\epsilon_i = 100$, $\Delta\eta_i = 190$ and $J_i = 16$ (the same as that shown in Figure 4(b)). The plot shows that the generated inter-arrival times are distributed under the desired Laplace distribution indicated by the dash line.

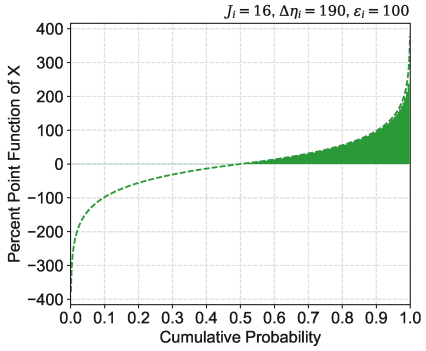


Figure 12: Chart of the percent of function (PPF) based on a Laplace distribution with $\epsilon_i = 100$, $\Delta\eta_i = 190$ and $J_i = 16$ (the same as that shown in Figure 4(b)). The dash line represents the true PPF curve and the bars are reconstructed by the 100 PPF points stored in the PPF-based distribution array converted using Algorithm 1.

B A LAPLACE RNG IN LINUX KERNEL

To create a Laplace distribution-based random number generator in the Linux kernel, we convert the distribution's PPF into an array to store in the kernel code by using Algorithm 1. This algorithm takes as input a function of PPF of the target distribution (centered at 0) and the desired number of the points (*steps*) to convert into an integer array as the output ($array_{PPF}$). In this algorithm, the PPF function takes as input a percentile value (ranged from 0 to 1.0) and gives the corresponding distribution sample value at the given percentile. An example of the PPF function is provided in Figure 12 as the dash curve. Line 3 computes the resolution of the percentage each point in the array represents. Line4 to line 8 iterate through each of the computed percentile to obtain and store the corresponding percent point value in the output array. Line 9 returns the array which stores PPF points above the 50-th percentile. In other words, the array contains only half part of the distribution (as demonstrated by the bars shown in Figure 12). It is done to save memory space as a Laplace distribution is symmetric. We then use Algorithm 2 to obtain a random number from the PPF array.

Algorithm 1: PPF-Based Distribution and Array Conversion

Input:
PPF =: the PPF of the target Laplace distribution
steps =: the number of PPF points to expand
Output:
$array_{PPF}$ the array storing the PPF points

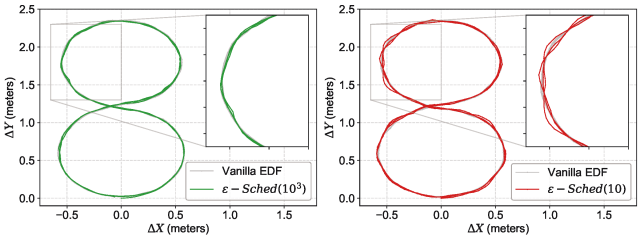
1 $array_{PPF} = []$
2 $step = 0$
3 $resolution = (1 - 0.5)/(steps - 1)$
4 **while** $step < steps$ **do**
5 $\quad percentile = step \cdot resolution + 0.5$
6 $\quad array_{PPF}[step] = int(PPF(percentile))$
7 $\quad step = step + 1$
8 **return** $array_{PPF}$

Algorithm 2: PPF-Based Random Number Generator

Input:
$array_{PPF}$ =: an array storing expanded PPF points
Output:
sample =: a random value equivalent to the corresponding distribution

1 $size_{array} = len(array_{PPF})$
2 $rad_{idx} = RAND_{int}(0, len(array_{PPF} \cdot 2 - 1))$
3 **if** $rad_{idx} > (size_{array} - 1)$ **then**
4 $\quad sample = -array_{PPF}[rad_{idx} - size_{array}]$
5 **else**
6 $\quad sample = array_{PPF}[rad_{idx}]$
7 **return** *sample*

Algorithm 2 takes as input the aforementioned PPF array ($array_{PPF}$) and draw a random number that is equivalent to a random draw from the underlying distribution. Line 2 obtains a random number from a common random number generator (based on a uniform distribution) with a range of $[0, 2 \cdot len(array_{PPF}) - 1]$ (*i.e.,* two times of the length of the PPF array). Line 3 to line 7 convert the random number into a feasible index to obtain a sample value from the PPF array. If the random number is greater than the array's length, a negative sample value is generated. Otherwise a positive value is obtained and returned.



(a) ϵ-Scheduler ($\epsilon = 10^3$) (b) ϵ-Scheduler ($\epsilon = 10$)

Figure 13: The trajectory of the rover system steering through predefined way points with RoverBot running under Vanilla EDF and ϵ-Scheduler. The worst observed deviations are $0.024m$ and $0.038m$ in the cases of $\epsilon = 10^3$ and $\epsilon = 10$ respectively, compared to the trajectory of Vanilla EDF.

Table 7: K-S Test and Average Minimum L2 Distance

Way Points	Comparison	K-S	p-val	Min Dist (Meters)
Route "8"	$\epsilon = 10^3$, vanilla	0.015038	0.9999	0.009458
	$\epsilon = 10$, vanilla	0.016958	0.9999	0.009940
	$\epsilon = 10^3$, $\epsilon = 10$	0.013265	0.9999	N/A

C AUTONOMOUS ROVER SYSTEM

C.1 The Route "8" Test

With an experiment setup the same as introduced in Section 7.1, we conduct another set of tests with a closed loop route that has a shape "8". The results are shown in Figure 13 and Table 7 which suggest similar, promising performance outcomes. As the "8" route has more rounded turns, the worst observed deviations ($0.024m$ and $0.038m$ in the cases of $\epsilon = 10^3$ and $\epsilon = 10$ compared to Vanilla EDF) are generally smaller (*i.e.,* better) compared to the irregular route.

C.2 The Kolmogorov-smirnov (K-S) Test

We perform two types of K-S tests: the one sample test and the two sample test. The one sample test is used to determine whether the cumulative distributive function (CDF) of an observed random variable is identical to the CDF of a reference random variable, also known as the null hypothesis. Here, our observed random variable is the y-axis of the rover paths with ϵ-Scheduler ($\epsilon = 10^3$ or $\epsilon = 10$) and our reference random variable is the y-axis of the rover paths with Vanilla EDF. The two sample test is used to determine whether two independent samples are drawn from the same continuous distribution (null hypothesis). Hence, this test is used to compare rover paths with ϵ-Scheduler ($\epsilon = 10^3$) and ϵ-Scheduler ($\epsilon = 10$). If the K-S statistic value is small and the corresponding p-value large, then we cannot reject the null hypothesis. Instead, the null hypothesis is almost certainly true.

D VIDEO STREAMING EAVESDROPPING ATTACK

While streaming with DASH, each video segment is a certain segment length and quality level. This type of mechanism results in a distinct traffic pattern due to the segment-based transmission. This key property can be used to identify videos while streaming.

Table 8: Video Description

Video	Content	Resolution	FPS
1	lecture	640 X 352	30
2	movie	640 X 360	30
3	street	480 X 360	25
4	soccer	640 X 480	25
5	cartoon	640 X 480	25

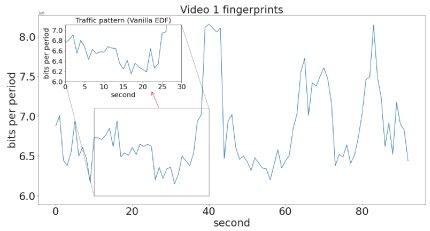


Figure 14: The similarity between the eavesdropped traffic pattern with Vanilla EDF when video 1 was being streamed and video 1's fingerprints. The similarity distance is calculated using temporal sequence analysis.

By eavesdropping on the network traffic during video streaming, attackers can recognize certain patterns in the traffic. Video fingerprints can be built on the other hand, using the pre-recorded video files. Hence, attackers can utilize the fingerprints and the observed traffic pattern to identify which video was being streamed during eavesdropping. The idea is to merely compare the extracted video fingerprints with the observed traffic pattern of the video stream to deduce an individual video. A matching method is necessary for effective outcomes post-comparison. In this way, the eavesdropped traffic and the video fingerprints provide seamless video identification.

Real-time video has many varying parameters which make it difficult to implement an efficient yet accurate attack. There are many quality levels of recorded video: some have noise, others are clear. Fingerprinting often is not effective when there is too much noise in a sample. This is because the fingerprints will not be unique and are rarely representative of the sample. The bandwidth plays a major role in the adaptive quality selection in the network. A high bandwidth automatically transmits samples using higher quality levels. This makes it problematic to observe a consistent pattern for the same video in the traffic trace. Another obstacle is the length of the eavesdropped sample. It is time-consuming to eavesdrop on the entire video. Also most times, the host doesn't play the entire video, so only a part of the video is present in the eavesdropped traffic. There is a workaround for this in DASH: the bitrate variation trend is stable for a particular sample, hence a bitrate based fingerprinting method is viable.

The systematic steps are: extracting the video fingerprints from pre-recorded videos, obtaining the eavesdropped traffic pattern, calculating a similarity estimate between the traffic pattern and the fingerprints using temporal sequence analysis (p-DTW) and finally, identifying the video using this similarity estimate. An example of a fingerprint and an observed traffic pattern is shown in Figure 14. Specifically, we use the fingerprint-based video identification attack detailed in sections IV and V by Gu *et al.* [25], adapting it to the video streaming application that we designed.

The fingerprints that we obtain follow the segmentation rules of DASH. Initially, we calculate the data per second in bits per second (bps) for each video in a differential manner in order to eliminate the impact of different quality levels on the fingerprints. This sequence of fingerprints, which correspond to data per second, need to be aggregated into segments before the matching step because DASH transmits video data in segments. Each segment covers a certain number of seconds; segment length is usually kept constant. Again a differential strategy is used to collect the data per segment and the resulting set is the set of video fingerprints available to the attacker. For each video in the dataset, a set of video fingerprints is calculated. Next comes the part where the attacker eavesdrops on the network traffic to obtain traffic traces. This occurs during the transmission of the video from the server to the client. Assuming that there are no other processes that require a large bandwidth, the attacker aggregates the obtained network traffic (in bps) into data per segment in a differential manner. The objective is to find a maximum match between the sequence of traffic traces and the sequence of individual video fingerprints. The video corresponding to the maximum match is identified to be the video that was being streamed during the eavesdropping period.

The similarity measurement is a method to find out which set of video fingerprints is likely to produce the extracted traffic pattern. Since the assumption is that only one video is streamed at a time, measuring the similarity of the pattern to each set of fingerprints will reveal the closest match. We treat this as a time series matching problem. Two important considerations before solving the problem have to be taken into account: eavesdropping can be short and the eavesdropped period may correspond to only a portion of the entire video. After normalizing the sequences using a sigmoid function, a newly proposed method called "partial dynamic time warping (P-DTW)" is used. There are several advantages of using this method over the classic DTW [6] method. Classic DTW tries to match the two sequences in their entirety, *i.e.,* using the full length of sequences to calculate the alignment cost. The series heads and tails are required to be matched. On the other hand, P-DTW tries to find the best local alignment between the two sequences, *i.e.,* it minimizes the distance between the traffic pattern and any proper sub-sequence of the fingerprints. The sub-sequence that results in this minimum is selected for calculating the final similarity between the fingerprint and the traffic traces. As eavesdropping occurs only for a part of the video, P-DTW is more suited to our attack. The similarity is quantified using the minimum distance (cost) of aligning the two sequences. Whichever set of video fingerprints renders the minimum distance is the identified video in our attack.

E GENERATION OF SIMULATION TASK SETS

A total of 6000 task sets are grouped by utilization from $\{[0.001+0.1\cdot x, 0.1+0.1\cdot x) \mid 0 \leq x \leq 9 \wedge x \in \mathbb{Z}\}$. Each group contains subgroups that have a fixed number of tasks from $\{5, 7, 9, 11, 13, 15\}$. A total of 100 task sets are generated for each of the 60 subgroups. The utilization for a task set is generated from a uniform distribution using the *UUniFast* algorithm [8]. Each task's period T_i is randomly drawn from $[10ms, 200ms]$ and the worst-case execution time C_i is computed based on the generated task utilization and period. The task phase is randomly selected from $[0, T_i)$.

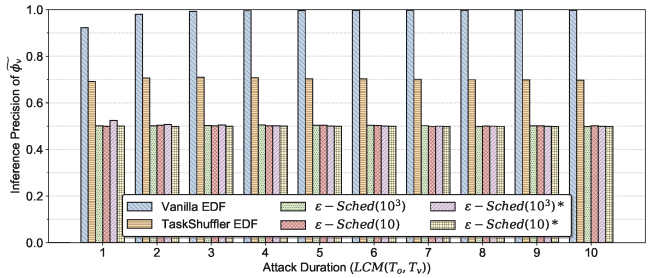


Figure 15: The inference precision results for a ScheduLeak attack duration ranged from $1 \cdot LCM(T_o, T_v)$ to $10 \cdot LCM(T_o, T_v)$. The experiment suggests that ϵ-Scheduler's protection against ScheduLeak is independent to the attack duration.

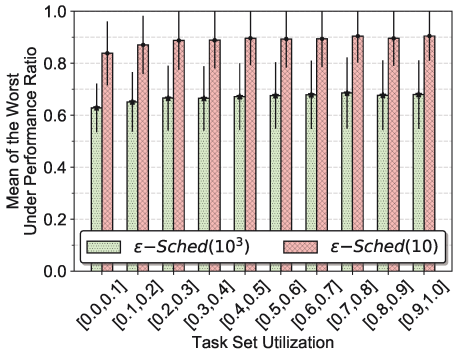


Figure 16: Results of the mean of the worst under-performance ratio (y-axis) grouped by the task set utilization (x-axis). The experiment gives us an insight into the degradation a system may observe from its tasks. It suggests that a task's under-performance ratio can be biased towards 0.5 and above. The bias is noticeable when ϵ is large. This often happens on the task that has a small period leading to an asymmetric distribution that tends to generate larger inter-arrival times.

F DISCRETE FOURIER TRANSFORM ANALYSIS SETUP

To adequately utilize such a tool, the task schedule must be transformed into a sequence of equal-spaced samples that represent the states when CPU is busy and idle. In our analysis, a sample is taken at each time tick and hence the Nyquist frequency is half of the tick rate. In contrast to prior work [42] where busy and idle states are translated into binary values 1 and 0, we translate them into 1.0 and -1.0 to reduce noise in the spectrum caused by the positive-biased sample values. The outcome of the transformation is a sequence of 1.0 and -1.0 numbers that is then analyzed using DFT. In the end, only the first half part of the analysis result is taken since the DFT output is known to be conjugate symmetric. As shown in Figure 7, the resulting frequency spectrum is useful for uncovering the periodicity introduced by the scheduling of the real-time tasks. Additionally, it can also be seen that peaks encapsulate the true frequencies of the tasks (annotated by the red dashed lines). It's worth noting that the spectrum can contain aliasing frequency peaks that

are in harmony with the true frequencies. These harmonic peaks in fact are helpful for adversaries to identify and verify the true frequencies of interest.

We are interested in the amount of information that an adversary can learn from the DFT analysis *w.r.t.* a task's periodic behavior. By the nature of DFT, the amplitude in the spectrum has a positive correlation with the periods and the peaks that stand out are particularly helpful to adversaries in gaining more knowledge about the schedule. To this end, we use a Z-score based peak detection algorithm [20, 38] to count the number of outstanding peaks in the spectrum. The peak detection algorithm uses a moving mean with a $10Hz$ window to detect the outstanding peaks that are 3.5 standard deviations away. As shown by the green line in Figure 7, such a moving threshold can effectively identify the peaks that are significant while filtering out the base noise.