# SCHEDULER SIDE-CHANNELS IN PREEMPTIVE REAL-TIME SYSTEMS: ATTACK AND DEFENSE TECHNIQUES

BY

CHIEN-YING CHEN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

## ABSTRACT

This dissertation aims to address the problem of the side-channels caused by the deterministic nature embedded in the real-time schedulers in preemptive real-time systems (RTS). The dissertation explores the problem by postulating that *there exist timing-based side-channels (i.e., scheduler side-channels) that enable adversaries to gauge the behavior of the system with high precision in preemptive RTS* and that *the RTS can be protected by diversifying the real-time schedules.* To validate this hypothesis, the work is divided into three groups to tackle the following three key challenges:

- Validate the presence of the scheduler side-channels in preemptive RTS.
- Protect the RTS by diversifying the real-time schedule.
- Evaluate the risks against the scheduler side-channels and the efficacy of a defense scheme.

The dissertation shows that the scheduler side-channels exist in both classes of widely used preemptive RTS (*i.e.,* fixed-priority RTS and dynamic-priority RTS) and can leak critical task information using a user-space, non-privileged task. Such information can be leveraged by other collaborative attacks (*e.g.,* advanced persistent threat attacks) to pose a serious threat to systems. A study on the schedule randomization technique as a defense strategy is conducted and shows that, while being effective in disturbing the repeated patterns in the schedule, there exist trade-offs (*e.g.,* the scheduling overhead) and shortcomings (*e.g.,* ineffectiveness in the face of real-time constraints.) Based on the lesson learned, the dissertation introduces the notion of "*schedule indistinguishability*" and presents a defense scheme that provides security guarantees to critical tasks by achieving the schedule indistinguishability. The scheduler relaxes the real-time constraints and add random noise drawn from bounded Laplace distribution to the task's execution patterns to hide the repeated patterns from the task schedule. The dissertation further introduces a security evaluation framework consisting of diverse metrics that capture the unique characteristics of real-time schedules and scheduler side-channels to better evaluate the risks for a given RTS. The work is concluded by assessing the developed scheduler against scheduler side-channels with using the introduced security evaluation framework.

*To my parents, for their love and support.*
*To my wife, for her unconditional commitment to this family.*
*To my daughter, for completing my life.*

## ACKNOWLEDGMENTS

I am grateful to have met many exceptional teachers, mentors, collaborators and friends on my path to finishing this dissertation. I would first like to thank my advisor Professor Sibin Mohan who guides me through my journey studying as a Ph.D. student. Your invaluable support encourages me to constantly challenge myself and explore all possibilities without fear of failures. This work could not have been done without your guidance. I would also like to express my appreciation on my doctoral committee, Professors Klara Nahrstedt, Nikita Borisov, Rakesh B. Bobba and Rodolfo Pellizzoni for their insightful comments and suggestions on improving the quality of the work. I express my special thanks to Professors Rakesh B. Bobba and Rodolfo Pellizzoni for pointing me in the right direction when I was in doubt tackling the research problems since the beginning of this work. Furthermore, the research work presented in this dissertation is not possible without the support from the National Science Foundation (NSF). I sincerely appreciate NSF for the opportunities.

I would like to extend my gratitude to my collaborators and friends, Dr. Yi-Zong Ou, Dr. Man-Ki Yoon, Monowar Hasan, Ashish Kashinath, Bin-Chou Kao, Hsuan-Chi Kuo, Kyo Hyun Kim, Chaitra Niddodi, Fardin Abdi, Rohan Tabish and Dr. Bo Liu for juicing up my journey. The days and nights we worked together will be the memories I remember at the University of Illinois Urbana-Champaign.

In addition, I would like to thank my former advisors and mentors, Professors Pai H. Chou, Ching-Lung Chang at National Tsing Hua University, Chuan-Yu Chang, Jenn-Kaie Lain, Po-Hui Yang at National Yunlin University of Science and Technology and my former supervisor Hanks Wu at HTC for establishing a rigid skill foundation that enables me to finish this dissertation. Your support gave me courage to pursue the doctoral degree at the University of Illinois Urbana-Champaign.

Last but not least, I would like to thank my parents and family for their love and support. They are always behind me with their beliefs in me. In particular, my wife has been giving her unconditional commitment to this family in support of the things I enjoy doing. My Ph.D. is not possible without you. I sincerely thank you for being with me throughout my adventure.

# TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.1 Research Goal and Challenges . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Summary of Solutions . . . . . . . . . . . . . . . . . . . . . . . . . . 3

CHAPTER 2 BACKGROUND AND RELATED WORK . . . . . . . . . . . . . . . 5
2.1 Real-Time Systems and Security . . . . . . . . . . . . . . . . . . . . . 5
2.2 Schedule Obfuscation . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 Differential Privacy and Randomized Mechanisms . . . . . . . . . . . . . 6

CHAPTER 3 EXPLOITATION OF SCHEDULER SIDE-CHANNELS . . . . . . . 9
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.2 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.3 The ScheduLeak Attack Algorithms . . . . . . . . . . . . . . . . . . . 12
3.4 The DyPS Attack Algorithms . . . . . . . . . . . . . . . . . . . . . . 24
3.5 Implementation in Real-Time Linux . . . . . . . . . . . . . . . . . . . 36
3.6 Evaluation Metrics and Setup . . . . . . . . . . . . . . . . . . . . . . 38
3.7 Evaluation Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.8 Case Studies on Real Platforms . . . . . . . . . . . . . . . . . . . . . 51
3.9 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
3.10 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

CHAPTER 4 DIVERSIFICATION OF REAL-TIME SCHEDULES . . . . . . . . . 57
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.2 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.3 The REORDER Scheduler . . . . . . . . . . . . . . . . . . . . . . . . 58
4.4 The $\epsilon$-Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.5 Implementation in Real-Time Linux . . . . . . . . . . . . . . . . . . . 76
4.6 Evaluation Metrics and Setup . . . . . . . . . . . . . . . . . . . . . . 81
4.7 Evaluation Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
4.8 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
4.9 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97

CHAPTER 5 ANALYTIC AND EVALUATION FRAMEWORK . . . . . . . . . . 98
5.1 Degree of Indistinguishability . . . . . . . . . . . . . . . . . . . . . . 98
5.2 Degree of Protection against Attacks . . . . . . . . . . . . . . . . . . 99
5.3 Quality-of-Service (QoS) . . . . . . . . . . . . . . . . . . . . . . . . 99
5.4 Usage of the Framework . . . . . . . . . . . . . . . . . . . . . . . . 99

CHAPTER 6 CONCLUSION . . . . . . . . . . . . . . . . . . . . . . . . . . 101

REFERENCES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102

# CHAPTER 1: INTRODUCTION

Real-time systems (RTS) such as autonomous cars, medicine/vaccine delivery drones, space rovers (*e.g.,* NASA's Opportunity and Spirit), industrial robots, implanted medical devices and power grid components, play a vital role in shaping today's technological evolution from everyday living to space exploration. In such systems, tasks delivering critical functionality rely on an operating system (typically a real-time operating system or 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.,* PID control processes, sensor data collectors, motor actuators, *etc.* as exemplified in the system shown in Figure 1.1) are designed as real-time tasks and executed in a periodic fashion to guarantee responsiveness.

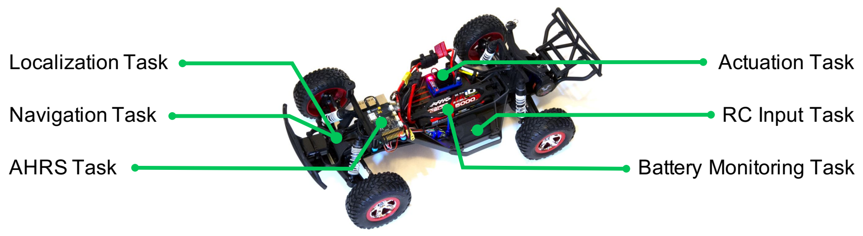
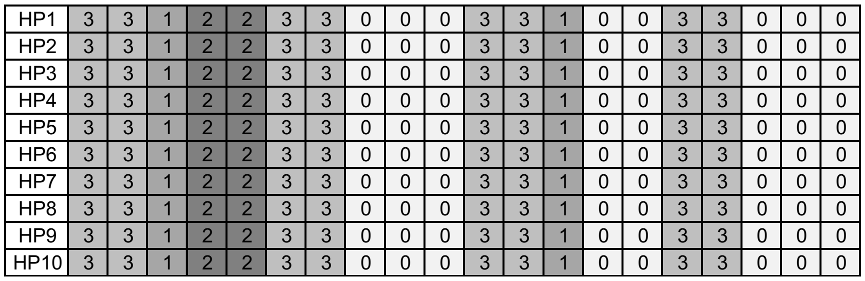


Figure 1.1: A rover system that exemplifies a typical real-time system. This system consists of six real-time periodic tasks that support an autopilot feature.

Due to the increased deployment of safety-critical systems that posses time-sensitive requirements in day-to-day environment to provide a broad range of services, security in real-time systems (RTS) is getting more focus in recent years. Traditionally, security has been an afterthought in the design of RTS, but the situation has changed with the use of commodity off-the-shelf (COTS) components. As exhibited by an increased number of security incidents [1, 2, 3, 4, 5, 6, 7, 8], RTS are undoubtedly becoming prone to attacks nowadays. Consequently, safety of RTS is at stake and understanding security threats they face is becoming more important every day.

To serve their safety-critical nature, RTS are designed with significant engineering effort to operate in a *predictable* manner. For instance, *(a)* designers take great care to ensure that the constituent tasks in such systems execute in an expected manner [9], *(b)* their interrupts are carefully managed [10] *(c)* the memory management is deterministic [11] and *(d)* the execution time is analyzed to great degree at compile time and run-time (*e.g.,* [12, 13, 14]). However, this predictability can be a double edged sword.

The predictability may introduce a timing-based side-channel in the system. A side-



| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| HP1 | 3 | 3 | 1 | 2 | 2 | 3 | 3 | 0 | 0 | 0 | 3 | 3 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 0 |
| HP2 | 3 | 3 | 1 | 2 | 2 | 3 | 3 | 0 | 0 | 0 | 3 | 3 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 0 |
| HP3 | 3 | 3 | 1 | 2 | 2 | 3 | 3 | 0 | 0 | 0 | 3 | 3 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 0 |
| HP4 | 3 | 3 | 1 | 2 | 2 | 3 | 3 | 0 | 0 | 0 | 3 | 3 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 0 |
| HP5 | 3 | 3 | 1 | 2 | 2 | 3 | 3 | 0 | 0 | 0 | 3 | 3 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 0 |
| HP6 | 3 | 3 | 1 | 2 | 2 | 3 | 3 | 0 | 0 | 0 | 3 | 3 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 0 |
| HP7 | 3 | 3 | 1 | 2 | 2 | 3 | 3 | 0 | 0 | 0 | 3 | 3 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 0 |
| HP8 | 3 | 3 | 1 | 2 | 2 | 3 | 3 | 0 | 0 | 0 | 3 | 3 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 0 |
| HP9 | 3 | 3 | 1 | 2 | 2 | 3 | 3 | 0 | 0 | 0 | 3 | 3 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 0 |
| HP10 | 3 | 3 | 1 | 2 | 2 | 3 | 3 | 0 | 0 | 0 | 3 | 3 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 0 |

Figure 1.2: An example of the real-time schedule produced by a vanilla EDF scheduler. It displays the schedules within a duration of 10 hyper-periods and shows that all hyper-periods have identical execution patterns due to the determinism in RTS.

channel allows an attacker to learn valuable knowledge about the target system from its implementation or behavior (*e.g.,* power consumption [15], electromagnetic emanations [16] and temperature [17].) To illustrate, Figure 1.2 shows the schedules of 10 hyper-periods that are generated from a RTS consisting of three periodic real-time tasks. As displayed in the figure, the execution pattern in the real-time schedule repeats after a hyper-period due to the deterministic nature of the RTS, thus the schedule is highly predictable. It enables adversaries to gauge the behavior of the system with high precision – I call such a vulnerability the "*scheduler side-channels*". Information leaked by using such side-channels can increase the success rates for certain attacks such as side-channel [18] and covert-channel attacks [19]. To gain comprehensive understanding for the cause and impacts of the problem, studies are conducted from both attacker's and defender's perspectives in this dissertation: *(i)* as an adversary, I identify the scheduler side-channels in two classes of preemptive RTS (*i.e.,* the fixed-priority and dynamic-priority preemptive RTS) and showcase possible consequences that result from a scheduler side-channel attack; *(ii)* as a defender, I present techniques to increase the difficulty of exploiting the scheduler side-channels in both classes. What follows presents the dissertation's core problem and summary of the solutions.

## 1.1 RESEARCH GOAL AND CHALLENGES

The work in this dissertation starts off by making the following hypothesis:

*There exist scheduler side-channels in preemptive RTS that can be defended against by diversifying the real-time schedules.*

Figure 1.3: The hypothesis made for this dissertation.

The objectives of this thesis are to provide an insight into the risks of scheduler side-channels and to offer a solution to protecting the critical components in RTS against such side-channels. The key questions this dissertation aims to answer are:

1. How does an adversary attack the periodic (and critical) components and, thus, validate the presence of the scheduler side-channels in preemptive RTS?
2. How does a system designer protect such critical components say, by diversifying the real-time schedule?
3. What are metrics that can be used to evaluate the risk posed by scheduler side-channels and the efficacy of defense schemes?

The challenges to answering these questions reside in the timing constraints imposed on RTS. While the adversaries may exploit the deterministic characteristics to gauge the behavior of the system via scheduler side-channels, the extraction of the run-time information is non-trivial. Main reasons include *(a)* the run-time schedule depends heavily on the state of the system at startup, initialization variables and environmental conditions and *(b)* there exist other real-time tasks (other than the attacker and the task being targeted) that can influence the task schedule. Even precise knowledge of all statically-known system parameters is insufficient to reconstruct the future execution behavior of the victim as its arrival times and start times are still unknown. While a privileged attacker could target the scheduler of the system and extract the requisite information, such access typically requires significant effort and/or resources. Therefore, there is a need to recognize the means for exploiting the predictable nature to leak a task's execution behavior.

On the other hand, a countermeasure that can protect the system from such side-channel attacks is in need. However, as demonstrated in this work, a scheduler side-channel attack requires only the minimum privilege to carry out (*i.e.,* a user-space task.) This makes it challenging (and possibly unrealistic) to completely eliminate scheduler side-channels. Rather than trying to redesign the entire system, this dissertation explores countermeasures on the real-time scheduler itself as it is where scheduler side-channels originate. To this end, the defender's limitations (*e.g.,* negative effects on system performance, scheduling constraints) must be identified for developing an effective defense approach.

## 1.2 SUMMARY OF SOLUTIONS

Based on the aforementioned challenges, the work is divided into three parts that are introduced in Chapters 3, 4 and 5.

First, to understand the scope of the problem, the dissertation studies the task relationship and the real-time schedules in the fixed-priority preemptive RTS and discovers the presence of scheduler side-channels. I develop the ScheduLeak attack algorithms [20, 21] that enable an unprivileged, low-priority task (that I call the "observer task") to learn the precise timing behavior of critical, periodic tasks (that I call the "victim tasks") by simply observing its own execution intervals and using a system timer. Based on ScheduLeak, similar attack algorithms – the DyPS algorithms [22], are developed to demonstrate the presence of the scheduler side-channels in dynamic-priority preemptive RTS where task priority relations between the observer task and the victim task does not persist. These results suggest that the scheduler side-channels can be exploited to infer the victim task's execution information with a high precision.

Second, with the knowledge obtained from the above attacks, I study a scheduling-based defense approach that utilizes a randomization techniques to obfuscate task schedules and disrupt the predictability of the real-time schedules. My experiments suggest that the proposed schedule randomization approach can effectively break the determinism in the real-time schedule. However, the results also show some shortcomings: *(i)* invoking a randomization function in the scheduler can increase the overhead of a context switch; *(ii)* obfuscating the task schedule is ineffective when the system utilization is high and, hence, the protection of critical task(s) is not guaranteed.

To address these issues, I introduce the notion of "*schedule indistinguishability*" and present a real-time scheduler that provides concrete security guarantees to critical tasks by achieving system indistinguishability. The proposed schedule indistinguishability is inspired by differential privacy [23, 24] in which random noise is added to protect data privacy in the context of statistical queries on databases. The scheduler relaxes the real-time constraints and add random noise drawn from bounded Laplace distribution to the task's execution patterns to hide the repeated patterns from the task schedules.

Next, to systematically evaluate the defense schemes as well as the scheduler side-channels, I introduce a security evaluation framework consisting of diverse metrics that capture the unique characteristics of real-time schedules and scheduler side-channels. This framework can serve as a common tool to evaluate the degree of protection offered by a defense scheme against the scheduler side-channels. The schedulers developed as part of this research ($\epsilon$-Scheduler, Chapter 4) are thoroughly assessed by using said security evaluation framework.

# CHAPTER 2: BACKGROUND AND RELATED WORK

## 2.1 REAL-TIME SYSTEMS AND SECURITY

Real-time tasks are usually constrained by their predefined minimum inter-arrival times (*i.e.,* periods), deadlines, the worst-case execution times (WCET) and real-time scheduling algorithms (*e.g.,* fixed-priority preemptive scheduling, earliest deadline first scheduling [9]). These real-time constraints help system designers analyze the system as a whole and ensure that all safety guarantees are met (*e.g.,* no real-time tasks will miss their deadlines). As a result, the system schedule (even a mix of both real-time and non-real-time tasks) becomes deterministic and highly predicable.

There has been some work on security in RTS [25, 26, 27, 28, 29, 30]. Most of the work focused on defence techniques against general attacks. Some researchers framed security in RTS as a scheduling problem [31, 32]. Hasan *et al.* [33] discussed the considerations of scheduling security tasks in legacy RTS. This is further extended to integrating security tasks into more general RTS models [34, 35]. There exist another area of work focused on hardening RTS from the architecture perspective. Mohan *et al.* [36] proposed to use a disjoint, trusted hardware component (*i.e.,* FPGA) to monitor the behavior of a real-time program running on an untrustworthy RTS. Yoon *et al.* [37] created the SecureCore framework that utilizes one of the cores in a multi-core processor as a trusted entity to carry out various security checks for the activities observed from other cores. Abdi *et al.* [38, 39] developed a restart-based approach that uses a root-of-trust (*i.e.,* a piece of hardware circuit) and the trust zone technology to enforce the system reboot process to evict any malicious dwellers when necessary. While some of the techniques are useful for detecting anomalies and mitigating the impact of the attacks, they do not protect RTS from (scheduler) side-channels.

The determinism and predictability, though favorable for the system analysis and safety, is a double-edged sword – they create side-channels in RTS. There has been some work (*e.g.,* [19, 40, 41, 42, 43, 44, 45]) studying and demonstrating the existence of side-channels and covert-channels as consequences of the determinism in RTS. In this dissertation, I'm particularly interested in the scheduler side-channels that leak system timing behavior via task schedules. The notion of the scheduler side-channels has been studied between two network users with a shared traffic scheduler based on the first-come-first-serve scheduling policy [46, 47]. In the RTS domain, my work [20] first introduced the scheduler side-channels in fixed-priority preemptive RTS and proposed the ScheduLeak algorithms that can extract execution behavior of critical real-time tasks from an observed task schedule at run-time. The work demonstrated that the leaked task information broadens the attack paths in RTS and enables further advanced attacks such as overriding control signals in cyber-physical systems and increasing accuracy of cache-timing side-channel attacks. Liu *et al.* [48] targeted the same attack surface (*i.e.,* the task schedule) and showed that precise period values of critical real-time tasks can be uncovered by using frequency spectrum analysis (*e.g.,* Discrete Fourier Transform, DFT, analysis). Such period information, while seemingly subtle, is a crucial stepping stone to the aforementioned and many attacks against RTS. The work also suggested that the analysis is feasible against the task schedules that are collected either internally or externally. Consequently, existing side-channels such as power consumption traces [15], schedule preemptions [20, 44], electromagnetic (EM) emanations [16] and temperature [17], *etc.*, that are useful in monitoring the system status may be seen against RTS.

## 2.2 SCHEDULE OBFUSCATION

Collaborative work with Yoon [49] attempted to close the scheduler side-channels by introducing a randomization scheduling algorithm that obfuscates the task schedules in the fixed-priority preemptive RTS. This idea is then extended to multi-core environment [50]. Similarly, Krüger *et al.* [51] developed a combined online/offline randomization scheme to reduce determinisms for time-triggered systems. Nasri *et al.* [52] conducted a comprehensive study on the schedule randomization approach and argued that such techniques can expose the fixed-priority preemptive RTS to more risks. While these work are centered on the problem of scheduler side-channels, they do not provide analytical guarantee for the protection against scheduler side-channels. Additionally, the above works target hard RTS that are highly constrained by strict real-time constraints and hence the effectiveness is limited by the real-time nature (as discussed in Section 4.6.2). In contrast, in Section 4.4, I focus on a more realistic RTS model that has flexible and more tolerable timing requirements. This enables me to explore more aggressive defense strategy to achieve higher (and analyzable) protection against the threats imposed by the scheduler side-channels.

## 2.3 DIFFERENTIAL PRIVACY AND RANDOMIZED MECHANISMS

### 2.3.1 Differential Privacy.

Differential privacy, along with the theorems and algorithms that build the foundation for protecting data privacy, was introduced by Dwork [23, 24] originally in the context of statistical queries on databases. It offers that an adversary looking at the output from any of the differentially private query mechanisms cannot reason with high confidence about the individual's data and, most importantly, such protection is quantifiable based on the foundation of randomized mechanisms (*e.g.,* Laplace distribution for drawing noise added to the output). It can be seen that differential privacy is used in many subjects addressing the issue of data privacy [23, 53]. There is also a growing trend to extend such a notion in the systems domain [54, 55, 56] to protect data privacy distributed among a group of devices/entities.

While in this dissertation I 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 the context of real-time schedules, I define the notion of task/job indistinguishability that depicts the probability to distinguish the execution states from one task/job to another in task schedules. Roughly speaking, a low indistinguishability enables an adversary to extract a task's execution from an observed task schedule with a high confidence and hence the system is prone to compromises via scheduler side-channels. To address such a problem, I propose the $\epsilon$-*Scheduler* (Section 4.4) that offers $\epsilon$-*indistinguishability* at a job level and/or a task level, subject to the system constraints as well as the system designer's security goal. This is achieved by embedding a randomized scheduling mechanism for adding noise to the inter-arrival times for each job at every scheduling point to abate the predictability and determinism. To the best of our knowledge this dissertation is the first work that adopts the foundation of differential privacy in the design of schedulers and to address the security issues in RTS.

### 2.3.2 Laplace Mechanism.

The Laplace distribution has been used in the classic differential privacy problems for generating random noise to achieve desired privacy protection [24]. Conventionally, the Laplace distribution has a probability density function defined as:

$$\mathrm{Lap}(x \mid \mu, b) = \frac{1}{2\,\underset{\text{scale parameter}}{b}}\exp\left(-\frac{|x - \overset{\text{location parameter}}{\mu}|}{\underset{\text{scale parameter}}{b}}\right) \tag{2.1}$$

where $\mu$ is a location parameter and $b > 0$ is a scale parameter. The distribution has a mean equal to $\mu$ and a variance equal to $2b^2$. In $\epsilon$-Scheduler presented in this dissertation, I use 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 [57, 58]) achieving the same level of indistinguishability, using Laplace distribution allows me to reuse existing mathematical and algorithmic components with great theory foundation from the differential privacy domain.

# CHAPTER 3: EXPLOITATION OF SCHEDULER SIDE-CHANNELS

In this chapter, I intend to validate the hypothesis about the presence of the timing-based side-channels in preemptive RTS. I show that such side-channels can leak execution behavior of periodic tasks (*i.e.,* the victim task) to unprivileged, user-space tasks (*i.e.,* the observer task) due to the deterministic nature embedded in the real-time schedulers (and hence I call such side-channels the "*scheduler side-channels*"). The leakage allows the attacker to learn the victim task's phase (and hence being able to infer the future arrival time points) as a step of reconnaissance for other advanced attacks that aim to create more severe damage to the victim system. Therefore, while a scheduler side-channel attack itself may not cause much loss, greater threats are posed in the aftermath of the leakage.

## 3.1 INTRODUCTION

Consider the scenario where an adversary wants to attack an embedded real-time system (RTS) – parts of autonomous cars, industrial robots, anti-lock braking systems in modern cars, unmanned aerial vehicles (UAVs), power grid components, the NASA rovers, implanted medical devices, *etc.* These systems typically have limited memory and processing power, have very regimented designs (stringent timing constraints for instance) and any unexpected actions can be quickly thwarted. Therefore, the opportunity to either steal a critical piece of information or the ability to launch that attack which takes control of the system can be very limited. As a consequence, attacks on such systems require significant system specific information. This "information" can take many forms – from an understanding of the design of the system, to knowledge of the critical components (either software or hardware). The exact knowledge depends on the type of attack and the target component. For example, say, *(a)* to steal important information about when (and where) an on-board camera is used for reconnaissance or *(b)* to take control away from the ground operator of a remotely-controlled vehicle.

The one common underlying theme that pervades real-time systems (and something that a would-be attacker should definitely address) is the importance of *timing*. "Timing" includes: *(i)* when certain events occur, *(ii)* how often they occur, and, most importantly for this work, *(iii) when (and if) they will occur again in the future.* In fact, a number of critical software components in real-time systems are *periodic* in nature. As we shall see, these periodic tasks present themselves as prime targets for attackers.

So, how does one attack such systems, especially the periodic (and critical) components?

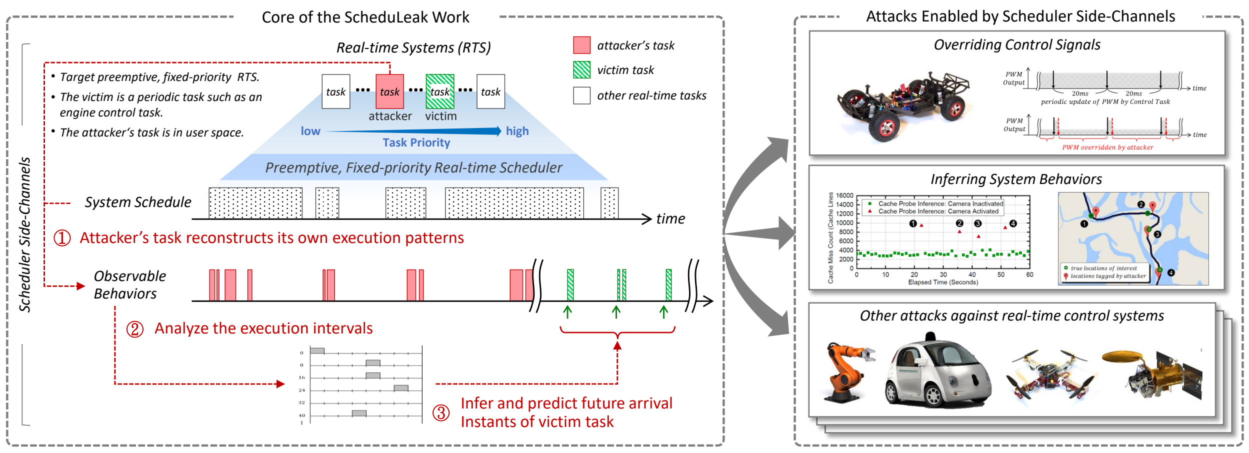

Core of the ScheduLeak Work
Real-time Systems (RTS)
Target preemptive, fixed-priority RTS.
The victim is a periodic task such as an engine control task.
The attacker's task is in user space.
task
attacker
victim
low
high
Task Priority
Preemptive, Fixed-priority Real-time Scheduler
attacker's task
victim task
other real-time tasks
Scheduler Side-Channels
System Schedule
time
① Attacker's task reconstructs its own execution patterns
Observable Behaviors
② Analyze the execution intervals
③ Infer and predict future arrival Instants of victim task
Attacks Enabled by Scheduler Side-Channels
Overriding Control Signals
PWM Output
periodic update of PWM by Control Task
PWM overridden by attacker
Inferring System Behaviors
Cache Probe Inference: Camera Inactivated
Cache Probe Inference: Camera Activated
Elapsed Time (Seconds)
true locations of interest
locations tagged by attacker
Other attacks against real-time control systems

Figure 3.1: Overview of the work (using fixed-priority RTS as an example): I demonstrate how an unprivileged, low-priority task (in user space) can use the ScheduLeak algorithms to infer execution behaviors of critical, high-priority periodic task(s). The extracted information is useful for helping other attacks achieve their primary goals (two such attack instances are implemented in this dissertation as possible use cases).

*I have discovered the presence of a scheduler-based side-channel that leaks timing information in real-time systems.*

The scheduler-based side-channel enables an *unprivileged, low-priority task* (in the case of fixed-priority RTS) to learn the precise timing behavior of the critical, periodic (victim) task(s) by simply observing its own execution intervals using a system timer. This provides an attacker with the ability to *infer the phase of the victim task and precisely predict its future arrival times at run-time.* I name the algorithms that exploit this side-channel attack "*ScheduLeak*" in fixed-priority RTS and "*DyPS*" in dynamic-priority RTS.

Figure 3.1 presents an overview of the side-channel and also how the attacker can benefit from the scheduler side-channel-based information. The left side of the figure shows how a real-time system consisting of real-time tasks (the boxes at the top – the victim is a periodic task while all other tasks can be either periodic or sporadic) that results in a schedule (dotted boxes in the middle, with each task being indistinguishable from the other at run-time) can be analyzed to extract the precise future arrival time points (the green, upward arrows) of the victim task. The right-hand side of the figure shows how this timing information of the critical task can be used to launch other attacks that either leak more important information or destabilize the real-time control system. Note that without this precise timing information, an attacker is either forced to guess when the victim task(s) will execute or launch the attacks at random points in time – both of which dilute the efficacy of the attack or result in early termination of the system.

The extraction of this run-time timing information is non-trivial; main reasons include *(a)* the run-time schedule depends heavily on the state of the system at startup, initialization variables and environmental conditions and *(b)* real-time systems typically include multiple non-real-time tasks as well. Even precise knowledge of *all* statically-known system parameters is insufficient to reconstruct the future arrival times of the victim. While a privileged attacker could target the scheduler of the system and extract the requisite information, such access typically requires significant effort and/or resources. On the other hand, the proposed algorithms are able to reconstruct the information with the same level of precision using an *unprivileged user space application.* This is achieved by letting the attacker's application keep track of its own scheduling information. Coupled with some easily obtainable information about the system (*e.g.,* the victim task's period), the attacker can recreate the targeting timing information with high precision.

The challenges can also be seen by examining some techniques that can potentially yield the same information. A possible strategy is to utilize Linux commands `ps` and `top`. However, these commands provide only basic process information (*e.g.,* priorities, runtimes) and offer coarser time resolution (in seconds). Hence, information gained from Linux commands is insufficient to determine the victim task's future execution time points. Another strategy is to employ Discrete Fourier Transform (DFT) that transforms a signals from the time to the frequency domains that seems to work at a first glance. However, unlike different signals accumulating as waveforms in the time domain, in a system schedule an execution interval is partitioned and deferred in the case of preemption – this distorts the original periodic signals in the frequency domain and hence makes the corresponding phase information unusable. Other factors that make DFT even more unfeasible include varying execution times and existence of other sporadic tasks.

To be more specific, let's say that we want to override the (remote) control of a rover. In many such systems, a periodic pulse width modulation (PWM) task drives the steering and throttle. Without knowledge of when the PWM task is likely to update the motor control values, the attacker is forced to employ brute force or random strategies to override the PWM values. These could either end up being ineffective or lead to the entire system being reset before the attack succeeds (see Section 3.8 for more details on this and another scenario). Armed with knowledge from ScheduLeak, our smart adversary can now override the PWM values *right after* they have been written by the corresponding task – effectively overriding the actuation commands.

Scheduler covert channels, where two processes covertly communicate using the scheduler, have long been known (*e.g.,* [19, 43, 44]). In contrast, the focus of this work is on a *side-channel that leaks execution timing behavior (not deliberately, as opposed to the scheduler* *covert channels) of critical, high-priority real-time tasks to unprivileged, low-priority tasks.* It is important for an attacker to *stay within the strict execution time budgets* allotted to the unprivileged task – especially during the phases when it is trying to observe and reconstruct the victim's timing behavior. Failing this requirement will likely cause other critical real-time functionality to fail or trigger a watchdog timeout that resets the system, leading to premature ejection of the attacker. This property is crucial during the 'reconnaissance' phase of what has come to be known as advanced persistent threat (APT) attacks [59, 60]. *E.g.*, it has been reported that attackers had penetrated and stayed resident undetected in the system for *months* before they initiated the actual attack in the case of Stuxnet [61]. Once they had enough information about system internals, they were able to craft effective attacks tailored to that particular system.

Next, I introduce the two attack algorithms, ScheduLeak and DyPS, that demonstrate the presence of the scheduler side-channels in the fixed-priority and the dynamic-priority preemptive RTS, respectively. Two attack cases that benefit from the inferences produced by the introduced attack algorithms are presented in 3.8.

## 3.2 PRELIMINARIES

In this work, I assume that the attacker has access to a system timer on the target system and therefore time measured by the attacker has the resolution equal to this system timer. The timer can be either a software or a hardware timer (*e.g.,* a 64-bit *Global Timer* in FreeRTOS or a `CLOCK_MONOTONIC`-based timer in Linux). I consider a discrete time model [62]. I assume that a unit of time equals a timer tick (of the timer that the attacker can access) and the tick count is an integer. All system and task parameters are multiples of a time tick. I 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 $[a, b-1]$.

## 3.3 THE SCHEDULEAK ATTACK ALGORITHMS

### 3.3.1 System and Adversary Model

I consider a uniprocessor (*i.e.,* single-core), fixed-priority, preemptive real-time system consisting of $n$ real-time tasks $\Gamma = \{\tau_1, ...\tau_n\}$. A task can be either a periodic or a sporadic task[1]. Each task $\tau_i$ is characterized by $(T_i, D_i, C_i, \phi_i, pri_i)$ where $T_i$ is the period (or the

[1]A task may also be an aperiodic task. However, in systems like real-time Linux (*i.e.,* Linux with the `PREEMPT_RT` patch), aperiodic tasks only get to run in slack time (*i.e.,* when no real-time tasks are in the

Table 3.1: A summary of the system and adversary model.

| **Real-Time System Assumptions** | |
|---|---|
| A1 | A preemptive, fixed-priority real-time scheduler is used. |
| A2 | The victim task is a periodic task. |
| **Attacker's Capabilities (Requirements)** | |
| R1 | The attacker has the control of one user-space task (observer task) that has a lower priority than the victim task. |
| R2 | The attacker has knowledge of the victim task's period. |
| R3 | The attacker has access to a system timer on the system. |
| **Attacker's Goals** | |
| G1 | Infer the victim task's phase and predict future arrivals. |

minimum inter-arrival time), $D_i$ is the relative deadline, $C_i$ is the worst-case execution time (WCET), $\phi_i$ is the task phase[2] and $pri_i$ is the priority. I assume that every task has a distinct period[3] and that a task's deadline is equal to its period [9]. I use the same symbol $\tau_i$ to represent a task's job (or instance) for simplicity of notation. I assume that task release jitter is negligible. Thus, any two adjacent arrivals of a periodic task $\tau_i$ has a constant distance $T_i$. I further assume that each task is assigned a distinct priority and that the taskset is schedulable by a fixed-priority, preemptive real-time scheduler. Let $hp(\tau_i)$ denote the set of tasks that have higher priorities than that of $\tau_i$ and $lp(\tau_i)$ denote the set of tasks that have lower priorities than $\tau_i$. I define an "execution interval" of a task to be an interval of time $[a, b)$ during which the task runs continuously. If $\tau_i$ is preempted then the execution will be partitioned into multiple execution intervals, each of which has length less than $e_i$.

I assume that an attacker is interested in targeting *one of the critical tasks in the system* that I henceforth refer to as a "*victim task*", denoted by $\tau_v \in \Gamma$. I also assume that $\tau_v$ is a *real-time, periodic* task. Many critical functions in real-time control systems are periodic in nature, *e.g.,* the code that controlled the frequency of the slave variable-frequency drives in the Stuxnet example [61]. In all such cases, the period of the task is strictly related to the characteristics of the physical system and thereby can be deduced from the physical properties; hence, I can assume that the attacker is able to gain knowledge of the victim

ready queue). As a result, aperiodic tasks do not influence how real-time tasks behave and thus are ignored in this dissertation.

[2]The task phase is defined as the offset from the zero time point to any of the task's arrival time points projected on the period on the zero time point. Thus, $\phi_i < T_i$. It should not be confused with the arrival time point of the task's first job.

[3]This assumption is in line with existing standards in the design of real-time tasks to ensure distinct periods/priorities. For example, AUTOSAR (a standardized automotive software architecture) tools map runnables/functions activated by the same period to a single task to reduce context switch/preemption overheads.

task's period beforehand. It is common that, before attacking complex systems (*e.g.,* CPS), adversaries will study the design and details of such systems. However, the attacker does not know the initial conditions at system start-up (*e.g.,* the task's phase) and may not have information on all the tasks in the system. All other tasks in the system can be either periodic, sporadic or non-real-time, depending on the design of the system. Hence, the methods developed in this work can target systems that have a mix of periodic, sporadic and non-real-time tasks.

The ultimate goal varies with adversaries and the systems under attack. For example, in advanced persistent threat (APT) attacks [59, 60], one may plan to interfere with the operations of critical tasks, eavesdrop upon certain information via shared resources or even carry out debilitating attacks at a critical juncture when the victim system is most vulnerable. Oftentimes, such attacks require the attacker to precisely gauge the timing properties of victim tasks. In this work, I introduce attack algorithms that help an attacker obtain this valuable information during the reconnaissance stage. In this context, the main goal of the attacker is to *precisely infer when the victim task is scheduled to run* in the near future (*i.e., the future arrival times*).

Note that my focus in this work is on how to reconstruct the timing behavior of a higher-priority periodic victim task using the scheduler side-channel without violating the real-time constraints. I do this from the vantage point of a compromised, lower-priority ("observer") task. I do not focus on *how* attackers get access to the observer task. They could use any number of known methods – from compromised insiders, to supply chain vulnerabilities in a multi-vendor development model (as is usually practiced for the design and development of large, complex systems such as aircraft, automobiles, industrial control systems, *etc.*) [32], to vulnerabilities in the software and network among others. Recent work has demonstrated that real-time systems like commercial drones contain design flaws and hence are vulnerable to compromise [63, 64]. The details of gaining access to an observer task are out of scope for this work. Nevertheless, it is important to note that the proposed attack algorithms *do not require the observer task to be a privileged task in the system.* A summary of assumptions, attacker's capabilities and goals is given in Table 3.1.

As previously mentioned, I refer to the lower-priority task that the attacker controls as an "*observer task*" and it is denoted by $\tau_o \in \Gamma$. It can be a user-space task. The only constraint I place on $\tau_o$ is that it has a lower priority than the victim task, $pri_v > pri_o$. The observer task can be either a periodic or a sporadic task and its period (or its minimum inter-arrival time) can be shorter or longer than the victim task. In particular, being a periodic task is a more restrictive condition since it reduces the flexibility available to an attacker (this will be clearer as I introduce the algorithms). That is, the case where a periodic observer task with a period $T_o$ and priority $pri_o$ can succeed, a sporadic observer task (by picking the same $T_o$ as the minimum inter-arrival time and the same priority $pri_o$) can also succeed. Therefore, when analyzing the attack capabilities in Section 3.3.6, I will consider a periodic observer task (or a sporadic observer task running at a constant inter-arrival time).

In this work, *I use the observer task to infer the victim task's phase $\phi_v$ that can be used to predict future arrivals of the victim task*. I let the observer task "monitor" its own execution intervals by using a system timer. Note that reading system time does not require privilege in most operating systems (*e.g.,* invoking `clock_gettime`() in Linux). The key idea here is that the intervals when the observer task is active *cannot contain the victim task's* execution or its arrival time point since the victim would have preempted the observer task. However, there are also other higher-priority tasks that can impact the observer task's execution behaviors. To the attacker, the challenge is to then filter out unnecessary information and extract the correct information about the victim task. This is explained in the following section.

### 3.3.2 Overview of the ScheduLeak Algorithms

In what follows I introduce the core algorithms of ScheduLeak. The main idea is that the victim task cannot run while the observer task is running since the latter has a lower priority. By reconstructing the observer task's own execution intervals and analyzing those intervals based on the victim task's period, I may infer the *initial offset* and *future arrival times* for the victim task. A high-level overview of the various analyses stages in the proposed ScheduLeak algorithms includes:

1. *Reconstruct execution intervals of the observer task*: first, the observer task uses a system timer to measure and reconstruct *its own* execution intervals (*i.e.,* times when it itself is active). [Section 3.3.3]

2. *Analyze the execution intervals*: The reconstructed execution intervals are organized in a "*schedule ladder diagram*" – a timeline that is divided into windows that match the period of the victim task. [Section 3.3.4]

3. *Infer the victim task's phase and future arrivals*: in the final step, the phase for the victim task is inferred. This information is then used to predict the future arrivals of the victim. Since the victim task is periodic in nature, the offset from the start of its own window translates to the offset from startup when the first instance of the victim task executed. [Section 3.3.5]

**Algorithm 3.1:** $\mathbb{E}(C_o, C'_o, \lambda)$ Reconstructing an Execution Interval

```
Input:
$C_o$: the worst case execution time of $\tau_o$
$C'_o$: remaining execution time of present job of $\tau_o$
$\lambda$: maximum reconstruction duration in a period
Output:
$t_{begin}, t_{end}$: start, end time of the detected interval
$C'_o$: updated remaining execution time of present job of $\tau_o$
1  $t_0 = ST$ // system timer
2  $t_{begin} = t_0$
3  $t_{stop} = t_{begin} + C'_o - (C_o - \lambda)$
4  $duration = 0$
5  while $duration \leq$ loop execution time unit and $t_0 < t_{stop}$ do
6      $t_{-1} = t_0$
7      $t_0 = ST$
8      $duration = t_0 - t_{-1}$
9  end
10 if $duration >$ loop execution time unit then
11     $t_{end} = t_{-1}$
12 else
13     $t_{end} = t_0$
14 end
15 $C'_o = C'_o - (t_{end} - t_{begin})$
16 return $\{t_{begin},\ t_{end},\ C'_o\}$
```

### 3.3.3 Reconstruction of Execution Intervals

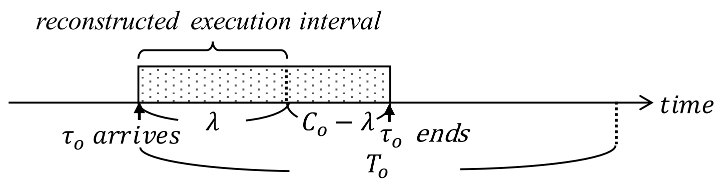
The first step is to reconstruct the observer task's execution intervals. I implement a function (Algorithm 3.1) in the observer task that keeps track of time read from the system timer. By examining the polled time stamps, preemptions (if any) can be identified and the execution intervals of the observer task can be reconstructed.

Algorithm 3.1 takes the observer task's worst case execution time $C_o$, the remaining execution time of the present instance $C'_o$ and the maximum reconstruction duration $\lambda$ as inputs. It outputs the start time $t_{begin}$ and end time $t_{end}$ of the detected execution interval as well as the updated remaining execution time of the present instance $C'_o$. *Lines 1 –4* initialize the variables to be used by the algorithm. Specifically, *line 3* computes the point in time (the stop condition) when the algorithm reaches the given maximum reconstruction duration $\lambda$ for the present instance. *Lines 5 – 9* are used to detect a preemption and check if current time exceeds the computed stop time point. These lines keep track of the time difference between each loop by reading present time from a system timer and comparing it to the time from the previous loop. If the time difference exceeds what I anticipate (the execution time of the loop), I know that a preemption occurred (*i.e.,* one or more higher-priority tasks executed). The loop exits either when a preemption is detected or the present time exceeds the computed stop time point. *Lines 10 – 12* determine the end time of the reconstructing execution interval. If the loop exits because of a preemption, the last time point before the preemption is taken as the end time of that execution interval (*line 11*). Otherwise, no preemption is detected, all $\lambda$ duration is used up and the latest time point is taken as the end time of the execution interval (*line 13*). *Line 15* updates the remaining execution time of the present job for the next invocation. *Line 16* returns the reconstructed execution interval and the updated remaining execution time.

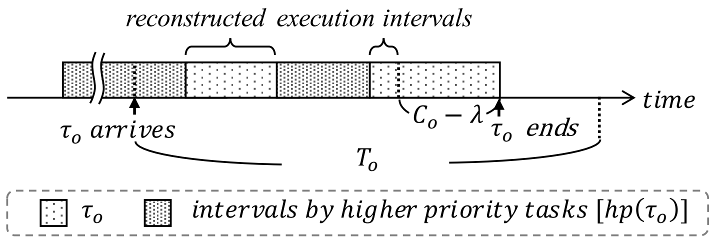
Algorithm 3.1 returns one execution interval of the observer task for every invocation. If required, the attacker can invoke this algorithm multiple times to reconstruct the execution intervals in detail. While this function seems straightforward, ensuring that it respects real-time constraints (*i.e.,* all real-time tasks must meet their deadlines) is critical. That is, the observer task should not run more than its WCET, $C_o$. Furthermore, even if the attacker does not exceed the allocated execution budget for itself, it may want to save some budget for other purposes such as performing the analyses to reconstruct the timing information of the victim. Hence, I define a parameter, $\lambda$, whose value is set by the attacker, to limit the running time of the aforementioned function for the observer task in each period. This "maximum reconstruction time", $\lambda$, is an integer in the range $0 \leq \lambda \leq C_o$. The total length of the reconstructed execution intervals is $\lambda$ in each period and this leaves the timespan $C_o - \lambda$ for the observer task to carry out other computations. As a result, the service levels guaranteed by the original (clean) system is still maintained – thus reducing the risk of triggering system errors. On the flip side, the attacker may not be able to capture all possible execution intervals and this could reduce the fidelity/precision of the final results. Section 3.3.7 discusses how to compute good values for $\lambda$. Figure 3.2 shows examples of reconstructed execution intervals.

### 3.3.4 Analysis of Execution Intervals

Once the observer task's execution intervals are reconstructed, I analyze the data to extract information about the victim task. I organize the observer's execution intervals into a timeline split into lengths of the victim task's period $T_v$ (recall that $T_v$ is one of the known quantities for the attacker). The purpose of this step is to place the execution intervals of the observer task within periodic windows of the victim task. The timeline split into windows of length that matches the victim task's period allows the attacker to see how the observer



(a) No preemption has occurred.



reconstructed execution intervals
time
$C_o - \lambda$
$\tau_o$ arrives
$\tau_o$ ends
$T_o$
$\tau_o$
intervals by higher priority tasks $[hp(\tau_o)]$

(b) Some tasks $\tau_i \in hp(\tau_o)$ preempt $\tau_o$.

Figure 3.2: Examples of reconstructed execution intervals of the observer task. The total length of the reconstructed execution interval(s) is $\lambda$ that leaves $C_o - \lambda$ for $\tau_o$ to perform original task functions.

task's execution intervals are influenced by the victim task as well as other higher-priority tasks.

To better illustrate the idea of the timeline and the proposed algorithms, I will use a "*schedule ladder diagram*" (defined below) to represent the construction of the timeline in this work. The rows in the schedule ladder diagram can be merged into a single-line timeline (and is just an analytical "trick"). A schedule ladder diagram is a skeleton consisting of a set of adjacent timelines of equal lengths – that match the victim task's period $T_v$. The start time of the top section can be an arbitrary point in time assigned by the attacker (*e.g.*, the time instant when the algorithms are first invoked). The columns in the schedule ladder diagram are "unit time columns". So, there are $T_v$ time columns. That is, the schedule ladder diagram has the same time resolution as the reconstructed execution intervals. The skeleton of a schedule ladder diagram is illustrated in Figure 3.3. From the diagram, plotted based on $T_v$, I make the following observation:

**Observation 3.1.** Any *schedule ladder diagram* of $\tau_v$ must contain exactly one arrival instance of $\tau_v$ in every row. All arrivals of $\tau_v$ are located in the same time column.

This observation is true because $\tau_v$ is a periodic task that arrives every $T_v$ time units and the schedule ladder diagram is plotted with its interval equal to $T_v$. I define the column where the arrivals of the victim task are located as the "true arrival column", denoted by $\delta_v$. Thus, the correlation between the initial offset $\phi_v$ and the true arrival column $\delta_v$ can

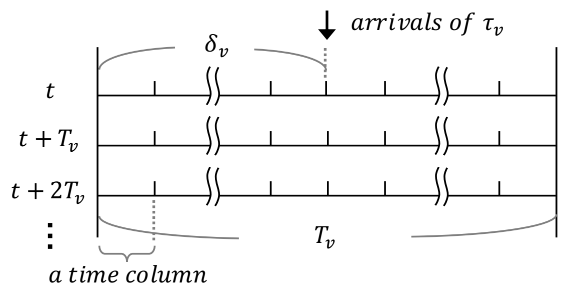


Figure 3.3: The skeleton of a schedule ladder diagram. It is used by the observer to organize its execution intervals. The start time $t$ of the schedule ladder diagram (*i.e.,* the beginning of the top timeline) is an arbitrary point in time, as assigned by the attacker. The width of each timeline matches the victim task's period $T_v$. The relative offset between the start time $t$ and the true arrival column is defined by $\delta_v$.

be derived by $(t + \delta_v - \phi_v) \ mod \ T_v = 0$, where $t$ represents the (arbitrary) start time of the schedule ladder diagram assigned by the attacker. This is also depicted in Figure 3.3. Based on this observation, I define the following theorem with respect to the observer task's executions on the schedule ladder diagram:

**Theorem 3.1.** The observer task's execution intervals do not appear at the time columns $[\delta_v, \delta_v + bcet_v)$, where $bcet_v$ is the best case execution time of $\tau_v$.

*Proof.* From Observation 3.1, the victim task $\tau_v$ arrives regularly at time column $\delta_v$. If there exists lower priority tasks $lp(\tau_v)$ in execution at $\delta_v$ column, the victim task preempts such tasks until it finishes its job with length of $bect_v$ at a minimum. In the case that there exists higher priority tasks $hp(\tau_v)$ that are executing or arriving during $[\delta_v, \delta_v + bcet_v)$, the victim task $\tau_v$ is preempted. Under this circumstance, if the observer task $\tau_o$ had arrived during $[\delta_v, \delta_v + bcet_v)$, as a lower priority task, it is also preempted. Therefore, the time columns $[\delta_v, \delta_v + bcet_v)$ *cannot* contain the execution intervals of the observer task. QED.

In other words, the time columns where the observer task $\tau_o$ can ever appear are not the true arrival column $\delta_v$. To this end, it's easier to think of the problem as the process of eliminating those such time columns. If I place the obtained execution intervals of $\tau_o$ on the schedule ladder diagram and remove the corresponding time columns, then, there must exist at least an interval of continuous time columns, of which the length is equal to or greater than $bcet_v$, that is not removed in the end. Those time columns are candidates for the true arrival time of $\tau_v$. There may also exist time columns that are not removed due to other higher-priority tasks. Yet, since other tasks have distinct arrival periods (or random arrivals for sporadic tasks), those time columns tend to be scattered (compared to $[\delta_v, \delta_v+bcet_v)$) and are expected to be eliminated as more execution intervals of the observer task are collected. In practice, the experiment results indicate that this process works effectively and is mostly stabilized after an attack duration of $5 \cdot LCM(T_o, T_v)$ (see Section 3.7.1).

**Example 3.1.** Consider an RTS consisting of four tasks $\Gamma = \{\tau_1, \tau_o, \tau_v, \tau_4\}$. For the sake of simplicity, I assume that all tasks are periodic in this example (though my analysis can work with periodic, sporadic and mixed systems as well). The task parameters are presented in the table below (on the left). Note that $pri_i > pri_j$ means that $\tau_i$ has a higher number than $\tau_j$. Thus, task $\tau_1$ has the lowest priority while task $\tau_4$ has the highest priority and $\tau_v$ has higher priority than $\tau_o$. Let the maximum reconstruction duration $\lambda$ be 1 and the start time of the attack be 0 (as a result, $\phi_v$ equals $\delta_v$ in this example). Assuming the attacker has executed the first step/algorithm for some duration, the table below lists the reconstructed execution intervals of the observer task.

Table 3.2: An RTS task set.

| | $T_i$ | $C_i$ | $\phi_i$ | $pri_i$ |
|---|---|---|---|---|
| $\tau_1$ | 15 | 1 | 3 | 1 |
| $\tau_o$ | 10 | 2 | 0 | 2 |
| $\tau_v$ | 8 | 2 | 1 | 3 |
| $\tau_4$ | 6 | 1 | 4 | 4 |

Table 3.3: Reconstructed execution intervals.

| Intervals |
|---|
| [0,1) |
| [12,13) |
| [20,21) |
| [30,31) |
| [43,44) |

Note that since $\tau_1$ has priority lower than the observer task $\tau_o$, it does not influence the execution of $\tau_o$. Then, I place the reconstructed execution intervals in a schedule ladder diagram of width equal to the victim task's period $T_v$. This operation is shown in Figure 3.5. To better understand the effectiveness of the schedule ladder diagram in profiling the victim task's behavior, I plot the original, complete, schedule on the ladder diagram in Figure 3.4 so that readers get a better sense of it. This gives us an insight into the relation between the execution intervals of $\tau_o$ and that of the victim task.

From the schedule ladder diagram in Figure 3.5, I remove the time columns that are occupied by the observed execution intervals. The results are shown at the bottom of Figure 3.5. What's left are candidate time columns that contain the true arrival times for the victim that I want to extract. These intervals are passed to the final step to infer the initial offset/arrival times of the victim task.

### 3.3.5 Inference of Initial Offset and Future Arrival Instants

I now get to the final step – inferring the future arrival instants of the victim task – our original objective. But, first, I need to calculate the initial offset of the victim task. What

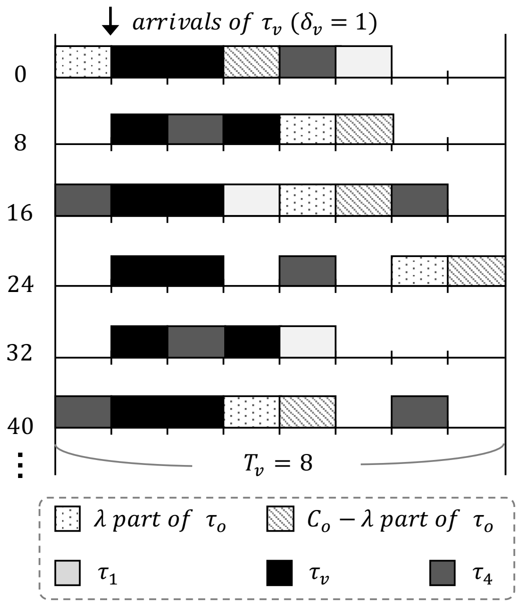
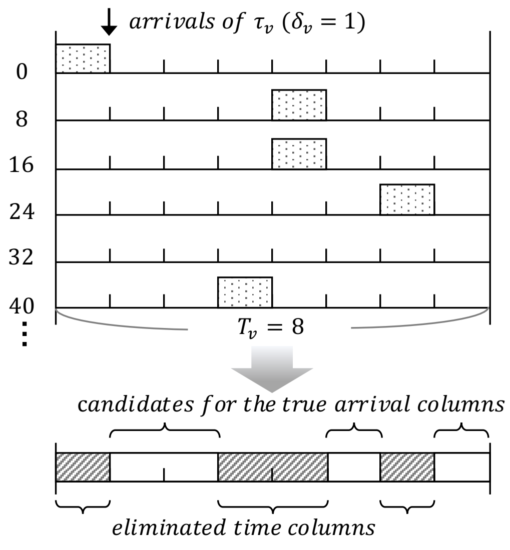

arrivals of $\tau_v$ ($\delta_v = 1$)
0
8
16
24
32
40
$T_v = 8$
$\lambda$ part of $\tau_o$
$C_o - \lambda$ part of $\tau_o$
$\tau_1$
$\tau_v$
$\tau_4$

Figure 3.4: The schedule of the task set in Example 3.1 plotted on a schedule ladder diagram with a width of $T_v$. It shows that time columns $[1, 3)$ are always occupied by either the victim task or other higher priority tasks. Therefore, the execution intervals of the observer task will not land on these time columns where the true arrival column is enclosed. This fact is what the proposed algorithms is based on.

I get from the previous step is a set of intervals of candidate time columns that contains the true arrival column of the victim task. The number of intervals depends on the number of collected execution intervals as well as the "noise" introduced by other, higher-priority, tasks (hence, there is no guarantee that all false time columns can be eliminated in the end). However, as observed from our experiments and based on Theorem 3.1, the false time columns tend to be scattered. Therefore, I take the largest interval as our inference that may contain the true arrival column of the victim task. I then pick the start of this interval as the inferred true arrival column, denoted by $\hat{\delta_v}$. While this strategy is not always guaranteed to succeed, the evaluation (both performance evaluation in Section 3.7 and case studies in Section 3.8) shows that the proposed algorithms are able to achieve a high degree of precision for the inference. The required initial offset, denoted by $\hat{\phi_v}$, can then be derived as $\hat{\phi_v} = (t + \hat{\delta_v})\ mod\ T_v$, where $t$ represents the start time of the schedule ladder diagram.

**Example 3.2.** The intervals obtained from Example 3.1 correspond to the time columns $[1, 3), [5, 6)$ and $[7, 8)$. According to the algorithm, the largest interval, $[1, 3)$, is selected. The starting point of such an interval is then taken as the inference of the victim task's true arrival column, which becomes $\hat{\delta_v} = 1$. In this example, the true arrival column is $\delta_v = 1$. Therefore, the algorithms correctly infer the true arrival column of the victim task and the



$arrivals\ of\ \tau_v\ (\delta_v = 1)$
0
8
16
24
32
40
$T_v = 8$
$candidates\ for\ the\ true\ arrival\ columns$
$eliminated\ time\ columns$

Figure 3.5: The processed schedule ladder diagram for Example 3.1.

initial offset can be derived accordingly.

Now, the future arrivals of the victim task can easily be computed by $\hat{\phi}_v + T_v \cdot \alpha$, $\alpha \in \mathbb{N}$, where $\hat{\phi}_v$ is the inferred initial offset of $\tau_v$, $T_v$ is the period of $\tau_v$ and $\alpha$ is the desired arrival number. The result of this calculation is the *exact time of the* $\alpha^{th}$ *arrival of the victim task.*

### 3.3.6 Analyzing Attack Capability

In this section, I discuss how to determine the attack capability or effectiveness of the observer task with respect to the victim task. That is, in this context, whether the observer task can remove all false time columns, and hence, correctly infer the arrival information of the victim task. Note that the analysis presented in this section focuses on the observer task being a periodic task since, as mentioned in Section 3.3.1, it is a more restrictive condition to an attacker. Given the same target system, a sporadic observer task may perform better as the sporadic task naturally has more flexible arrivals that are constrained only by its minimum inter-arrival time. A conservative condition ensuring that all false time columns can be removed from the schedule ladder diagram of $\tau_v$ is: when the observer task's execution intervals appear in all possible time columns. Therefore, I first analyze how the observer task's execution relates to the victim task's execution. When considering both $\tau_v$ and $\tau_o$ as periodic tasks, I have the following observation and theorem:

**Observation 3.2.** In the schedule ladder diagram, the *offset* between the time column of each observer task's arrival (*i.e.,* the scheduled execution) and the true arrival column repeats after their *least common multiple*, $LCM(T_o, T_v)$.

**Theorem 3.2.** If the given observer task $\tau_o$ and the victim task $\tau_v$ satisfy the inequality $C_o \geq GCD(T_o, T_v)$, then the scheduled execution of $\tau_o$ is guaranteed to appear in all time columns of the schedule ladder diagram of $\tau_v$.

*Proof.* From Observation 3.2, the time column offset of the observer task's execution repeats every $LCM(T_o, T_v)$. Therefore, the aforementioned condition (*i.e.*, the scheduled execution of $\tau_o$ appears in all possible time columns) can be described by the inequality $\frac{LCM(T_o,T_v)}{T_o} \cdot C_o \geq T_v$. Then, by using $LCM(T_o, T_v) = \frac{T_o T_v}{GCD(T_o,T_v)}$, I can derive a condition for $C_o$ that guarantees that the observer task can detect the arrivals of the victim task to be $C_o \geq GCD(T_o, T_v)$. QED.

From Theorem 3.2, I find that the observer task's scheduled execution can appear in some of the time columns more than once during $LCM(T_o, T_v)$ when $C_o > GCD(T_o, T_v)$. The redundant coverage means that the false time columns will be visited by $\tau_o$ more frequently when compared to the lower ratio of $C_o$ to $GCD(T_o, T_v)$. In contrast, if $C_o < GCD(T_o, T_v)$, then not all the false time columns can be covered and examined by the observer task. To better profile the observer task's coverage, I further define a *coverage ratio* that depicts the observer task's capability against the victim task as follows

**Definition 3.1.** (Coverage Ratio) The *coverage ratio*, denoted by $\mathbb{C}(\tau_o, \tau_v)$, is computed by

$$\mathbb{C}(\tau_o, \tau_v) = \frac{\overbrace{C_o}^{\text{viable observation length}}}{\underbrace{GCD(T_o, T_v)}_{\text{greatest common divisor of } T_o \text{ and } T_v}} \tag{3.1}$$

The *coverage ratio* can be loosely interpreted as the proportion of the time columns where the observer task can potentially appear in the *schedule ladder diagram*. If all $T_v$ time columns can be covered by the observer task, then $\mathbb{C}(\tau_o, \tau_v) \geq 1$. Otherwise $0 \leq \mathbb{C}(\tau_o, \tau_v) < 1$.

### 3.3.7 Choosing The Maximum Reconstruction Duration $\lambda$

Recall that, the maximum reconstruction duration $\lambda$ is used to limit the amount of execution time (in a period) taken up by the observer task for running the attack algorithms. As the attacker wants to stay stealthy and minimize disruption to the original functionality, it is desirable to use a $\lambda$ value as small as possible. The remaining execution time $C_o - \lambda$ can then be used by the attacker to deliver the original functionality of $\tau_o$ while making progress on the capturing of execution data. Based on this idea, $\lambda$ can be determined by:

$$\lambda = \begin{cases} GCD(T_o, T_v) & \text{if } \mathbb{C}(\tau_o, \tau_v) \geq 1 \\ C_o & \text{otherwise} \end{cases} \tag{3.2}$$

In the case of $\mathbb{C}(\tau_o, \tau_v) \geq 1$, the observer task has redundant coverage. Since a one-time coverage is sufficient for the observer task to examine all $T_v$ time columns, the additional coverage can be traded for other purposes. Otherwise ($\mathbb{C}(\tau_o, \tau_v) < 1$), the attacker may need to utilize all its computational resource for the attack.

## 3.4 THE DYPS ATTACK ALGORITHMS

One important challenge to leaking information via scheduler side-channels in dynamic-priority RTS, *e.g.,* the earliest-deadline first (EDF) algorithm, is that (relative) task priorities are not constant and vary at run-time. In the ScheduLeak attack that targets FP RTS, the observer task has a priority lower than the victim task at all times. This determinism ensures that the execution of the observer task is always preempted or delayed by the victim task when both are ready to run and this is vital for inferring the arrival times of the victim. In contrast, in an EDF RTS, task priorities are determined dynamically based on each job's absolute/relative deadline at each scheduling point. That is, no task will always have a higher priority relative to another task in the system. Hence, the ScheduLeak assumption about a persistent (relative) priority relationship between any two tasks in the system becomes invalid in EDF RTS.

### 3.4.1 System and Adversary Model

In this work, I consider a uni-processor, single-core, preemptive, dynamic-priority RTS running the EDF scheduling algorithm. The system consists of $n$ real-time tasks $\Gamma = \{\tau_1, \tau_2 ..., \tau_n\}$, each of which can be either a periodic or a sporadic task. A task $\tau_i$ is modeled by $C_i$, $T_i$, $D_i$, $\phi_i$ where $C_i$ is the worst-case execution time (WCET), $T_i$ is the period (or the minimum inter-arrival time for a sporadic task), $D_i$ is the relative deadline and $\phi_i$ is the task phase[4]. I assume that every task has a distinct period (or the minimum inter-arrival

[4]The task phase is defined as the offset from the zero time point to any of the task's arrival time points projected on the period on the zero time point. Thus, $\phi_i < T_i$. It should not be confused with the arrival time point of the task's first job.

time) and that $D_i = T_i$ [9]. I denote the $k$-th job of the task $\tau_i$ by $\tau_i^k$ and it is modeled by $c_i^k$, $a_i^k$, $s_i^k$, $d_i^k$ where $c_i^k$ denotes the execution time ($c_i^k \leq C_i$), $a_i^k$ is the absolute arrival time, $s_i^k$ is the start time and $d_i^k$ is the absolute deadline. For simplicity, I use $c_i$, $a_i$, $s_i$, $d_i$ when referring to an arbitrary job of $\tau_i$ if the job ordering is unimportant and I use "task" and "job" interchangeably. I only consider the task set that is *schedulable* by the EDF scheduling algorithm. Thus, $\sum_{\tau_i \in \Gamma} \frac{C_i}{T_i} \leq 1$. I assume that the task release jitter is negligible, and thus $a_i^{k+1} - a_i^k = T_i$ for a periodic task and $a_i^{k+1} - a_i^k \geq T_i$ for a sporadic task.

In EDF scheduling, a job with smaller absolute deadline gets to run first and is considered to have higher priority among other ready jobs that have greater absolute deadlines. In the case that multiple ready jobs have the same absolute deadline, they are considered to have the same priority and the EDF scheduler *randomly* selects one of the jobs to run. I define a task's "execution interval" to be an interval during which the task runs continuously.

Similar to the adversary model introduced in ScheduLeak, I assume that the attacker is interested in learning the task phase (and then inferring the future arrival time points) of a critical, *periodic* task (the victim task, denoted by $\tau_v$) in the system. This is considered to be part of a reconnaissance phase that can be a part of a larger attack. Such an attack will benefit from the inferences of the task's future arrival time points. The attacker launches the proposed DyPS attack algorithms that exploit the scheduler side-channels using an unprivileged, periodic task (the observer task, denoted by $\tau_o$) running on the same victim system. Once the inference of the victim task's phase is obtained, it is up to the attacker to decide if further attacks should be launched using the same observer task or via other attack surfaces. The ultimate goal of the attacks varies with the adversaries. Two examples of how the inferred information can be used to carry out further attacks are presented in Section 3.8.

The DyPS attack requires only the observer task to ensure the success for the attack algorithms introduced in this work. I assume that the observer task has access to a system timer that has a resolution that is coarser than or equal to a time tick. The observer task uses the timestamps read from such a system timer to reconstruct its own execution intervals and infer the victim task's phase. However, this method only works when the victim task has a priority higher than the observer task [20, Theorem 1], which is not always true under the EDF scheduling algorithm. More specifically, in the case of EDF RTS, the work is more interested in the priority relationships between tasks at *the instant when the victim task arrives.* To better clarify the relation between the observer task and the victim task, I define the term "observability" as follows:

**Definition 3.2.** (Observability) A victim task's arrival at $a_v$ is said to be observable by the observer task if the observer task has a priority lower than the victim task at $a_v$.

In an FP RTS, it is trivial to see that every arrival of the victim task is observable by the observer task if the victim task has a fixed, higher priority than the observer task. In an EDF RTS, it depends on both the tasks' periods and the absolute deadlines at run-time. I first determine the observability of a given observer and victim task pair by:

**Theorem 3.3.** For the EDF scheduling algorithm, given an observer task $\tau_o$ and a victim task $\tau_v$ whose periods are $T_o$ and $T_v$ respectively and $T_o \neq T_v$, the victim task's arrivals may be observable by the observer task only if $T_o > T_v$.

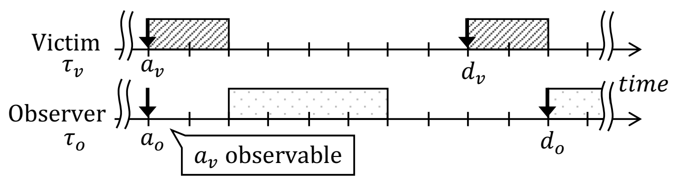
*Proof.* Definition 3.2 for EDF means that the observer task has a larger deadline when the victim task arrives. That is, $a_o \leq a_v$ and $d_o > d_v$ (or $a_o + T_o > a_v + T_v$) for some jobs of $\tau_o$ and $\tau_v$. Rewriting the above deadline inequality as $T_o - T_v > a_v - a_o$ indicates that $T_o$ must be greater than $T_v$ since $a_v - a_o \geq 0$. Now let's consider the case when $T_o < T_v$. It can be seen that when both tasks arrive at the same time (which is the case when the observer task has the largest possible deadline relative to the victim task), the observer task still has a deadline smaller than the victim task (*i.e.,* $a_o = a_v$ and thus $a_o + T_o < a_v + T_v \Rightarrow d_o < d_v$). Therefore, no victim task's arrival can be observed by the observer task if $T_o < T_v$. QED.

Based on Theorem 3.3, I make an assumption that the observer task must have a period larger than the victim task (*i.e.,* $T_o > T_v$) in this work.

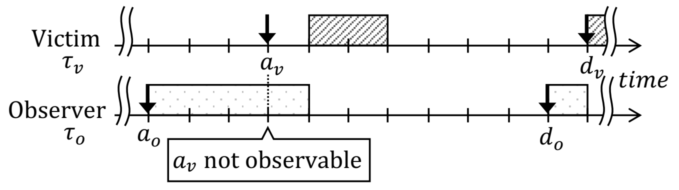
### 3.4.2 Challenges and Overview of the DyPS Algorithms

The scheduler side-channels in the FP RTS enable an unprivileged, low-priority task to learn precise timing information of a periodic, critical task. Similar scheduler side-channels exist in the EDF RTS due to the fact that both types of RTS are preemption-based systems. However, because of the dynamic nature, there are additional conditions and restrictions to be considered for making the attack succeed under the EDF scheduling algorithm. In this section, I present these constraints along with the details of the proposed DyPS algorithms.

In this work, the attacker's goal is to infer the victim task's phase and then predict its future arrival time points. This is achieved by allowing the observer task to reconstruct and analyze *its own execution intervals.* When the victim task arrives with a priority higher than the job of the observer task that has been scheduled, the execution of the latter is either delayed or preempted. As a result, the victim task's execution (including the arrival instant) is enclosed in the observer task's execution intervals. By reconstructing execution intervals for a sufficiently long duration (see Section 3.7.2 for the evaluation of the attack



(a) $d_v < d_o$ and thus $a_v$ is observable by $\tau_o$



(b) $d_v > d_o$ and thus $a_v$ is not observable by $\tau_o$

Figure 3.6: Examples of the two conditions (the victim task's arrival, $a_v$, being observable and not observable by the observer task under the EDF scheduling) elaborated in Example 3.3.

duration), it is possible to infer the victim task's phase. Yet, even if I assume that $T_o > T_v$ (Theorem 3.3) for a given observer and victim task pair, it is not guaranteed that every arrival of the victim task is observable by the observer task due to the dynamic priority in EDF.

**Example 3.3.** Consider an observer task $\tau_o$ ($T_o = 10$, $C_o = 4$) and a victim task $\tau_v$ ($T_v = 8$, $C_v = 2$). Without making any assumption on the task phases, Figure 3.6 demonstrates how dynamic priorities impact the observability in EDF schedules. In Figure 3.6(a), both $\tau_o$ and $\tau_v$ arrive at the same time point (*i.e.,* $a_o = a_v$). Since $\tau_v$ has a higher priority (because $d_v < d_o$), the execution of $\tau_v$ delays $\tau_o$ that is supposed to start at $a_o$. As a result, the arrival at $a_v$ can be observed by $\tau_o$. In Figure 3.6(b), $\tau_v$ arrives at a point when $d_o < d_v$. As $\tau_o$ is currently executing and has a higher priority at the arrival point $a_v$, the execution of $\tau_v$ is delayed by the execution of $\tau_o$. Consequently, the observer task fails to observe the victim task's arrival at $a_v$.

Apart from the fact that not all the observer task's execution intervals encapsulate the victim task's arrivals, the above example also hints at the fact that a part of an execution interval may still be admissible even when such an execution interval is considered invalid *w.r.t.* the observation of the victim task's arrivals. To illustrate, let's consider the observer task's execution interval starting at $a_o$ in Figure 3.6(b). It is in fact safe to leverage the first half of the execution interval to infer there is no arrival. This is because if the victim task arrived within this interval it would have higher priority (earlier deadline) and would preempt the observer task. But considering the second half of the execution interval to make the same inference would cause a false negative observation (*i.e.,* no arrival is inferred while there is one) and mislead the attack results. To identify the part of the execution interval that is valid for observing the victim task's arrivals, I present the following theorem:

**Theorem 3.4.** For a given job of the observer task arriving at $a_o$, only the execution interval(s) within $[a_o, a_o + T_o - T_v)$ is valid for observing the arrivals of the victim task.

*Proof.* The observer task may observe the victim task's arrivals if $a_o \leq a_v$ and $d_o > d_v$. However, it is unknown to the attacker when a job of the victim task would arrive, so the attacker must assume that the victim task may arrive at any time point and exclude the part of the execution intervals that may contain false information. For the jobs of $\tau_o$ whose deadlines satisfy $d_o > d_v$, all the execution intervals in such jobs provide valid observations of the victim task's arrivals. That is, for a given $a_o$ of a job whose deadline meets $d_o > d_v$, all the execution intervals within the range $[a_o, a_o + T_o)$ are valid.

In contrast, when $d_o \leq d_v$, the execution of $\tau_o$ may interfere[5] with the victim task's arrivals. From $\tau_o$'s point of view, the earliest victim task's arrival that may be interfered by $\tau_o$ occurs when $d_o = d_v$ and that arrival time point can be represented by $a_v = d_v - T_v = d_o - T_v = a_o + T_o - T_v$. Hence, it is possible for the observer task's execution to interfere with any arrivals of the victim task if it spans across the time point $a_o + T_o - T_v$. Therefore, only the execution intervals within $[a_o, a_o + T_o - T_v)$ is valid for the observer task to observe the victim task's arrivals. QED.

Using Theorem 3.4, it is possible for the observer task to reconstruct only the valid part of the execution intervals. However, the attacker may not directly use such a theorem as it requires $a_o$ (or more precisely, the task phase $\phi_o$) to be known. If the attacker is already present when the system starts, $\phi_o$ may be known to the attacker. However, in most attack cases where the attacker enters the victim system after the system starts, the attacker may not be able to easily learn the exact value of $\phi_o$ without further reconnaissance. In such cases, the attacker must first obtain $\phi_o$ before employing Theorem 3.4 and proceeding to infer the victim task's phase.

The rest of this section details every step of the proposed DyPS algorithms that account for the aforementioned challenges. An overview of the attack steps is given next:

1. Reconstruct $\phi_o$: the first step is to reconstruct the observer task's phase in order to identify the range specified in Theorem 3.4. [Section 3.4.3]

[5]By "$\tau_i$ interferes with an arrival of $\tau_j$" I mean that $\tau_i$ has a priority higher than $\tau_j$ at the arrival point of $\tau_j$ and hence the start of $\tau_j$ will be delayed by the execution of $\tau_i$.

2. Reconstruct execution intervals: with the reconstructed $\phi_o$, the observer task then is able to reconstruct only the valid part of the execution intervals for observing the victim task's arrivals. [Section 3.4.4]

These preceding two steps are introduced in DyPS for overcoming the challenges posed by the dynamic priorities in EDF. The information extracted using these two steps is compatible with the ScheduLeak algorithms. Hence, in this work I reuse existing work and toolboxes to carry out analysis. In particular, I leverage the following two parts of ScheduLeak:

1. Compute candidates: the observer task analyzes the reconstructed execution intervals and compute a list of time points as candidates for the final inference. [Section 3.4.5]

2. Infer $\phi_v$ and predict future $a_v$: a time point is selected from the candidate list as the inference of the victim task's phase. A future arrival time point of the victim task can then be predicted by using the inferred task phase. [Section 3.4.6]

### 3.4.3 Reconstructing The Observer Task's Phase

Reconstructing the observer task's phase, $\phi_o$, can be carried out by the observer task itself. When a new job of the observer task is scheduled to run, $a_o$ is unknown since there might be higher priority tasks delaying the observer task's execution. However, what the observer task itself can learn is the job's start time $s_i$ (by reading the time stamp right as the job starts) which is bounded by $a_o \leq s_o \leq d_o - C_o$. Let $\widetilde{\phi_o}$ be the reconstructed task phase of the observer task. For a given job, I may compute $\widetilde{\phi_o}$ from the start time $s_o$ by $\widetilde{\phi_o} = s_o \bmod T_o$ where $T_o$ is known to the attacker. Intuitively, the closer $s_o$ is to $a_o$, the more accurate $\widetilde{\phi_o}$ will be. By collecting and examining more start times, the attacker may further improve $\widetilde{\phi_o}$ by first determining the closest $s_o$ to $a_o$ and then computing $\widetilde{\phi_o}$ using the aforementioned equation. When there exists one job whose $s_o$ is equal to $a_o$, the correct task phase can be reconstructed (*i.e.*, $\widetilde{\phi_o} = \phi_o$). In Section 3.4.7 I discuss the factors that impact the reconstruction of the task phase and how likely it is for the attacker (observer task) to observe a situation where $s_o = a_o$ in a given task set. I now formally describe the algorithms.

Consider the observer task launching the attack on its $k$-th job ($k$ is an arbitrary number that is unknown to the attacker) and collecting its own job start times for $m$ jobs. What the observer task captures is a set of start times of consecutive jobs $S_o^{observed} = \{s_o^k, s_o^{k+1}, ..., s_o^{k+m-1}\}$. Our goal is to find the start time that is closest to its arrival time. I do this by comparing the start times using the following proposition:

**Proposition 3.1.** Given two start times $s_o^k$ and $s_o^{k+p}$ where $p \geq 1$ and thus $s_o^k < s_o^{k+p}$, I can determine the start time that is closer to its arrival time to be

$$\begin{cases} s_o^k & \text{if } s_o^k < s_o^{k+p} - p \cdot T_o \\ s_o^{k+p} & \text{otherwise.} \end{cases} \tag{3.3}$$

where $T_o$ and $p$ are known.

In the above proposition, the given start time pair, $s_o^k$ and $s_o^{k+p}$, represents two jobs differing in $p$ periods. Therefore, the start time of the $(k+p)$-th job can be shifted to the same period as the $k$-th job by $s_o^{k+p} - p \cdot T_o$ since the observer task arrives periodically (*i.e.*, $a_o^k = a_o^{k+p} - p \cdot T_o$). As a result, the two start times in the same period ($s_o^k$ and the shifted $s_o^{k+p} - p \cdot T_o$) are comparable. The smaller one is closer to its arrival time since $a_o \leq s_o$.

By employing Proposition 3.1, I can determine the start time that is closest to its arrival time using $S_o^{observed}$. The inference of the task phase for the observer task is computed by

$$\widetilde{\phi_o} = \min(s_o^{k+p} - p \cdot T_o \mid 0 \leq p < m) \bmod T_o \tag{3.4}$$

where $T_o$ is the period known to the attacker and $k$ can be unknown. Then, given a start time $s_o$, the attacker can compute its projected arrival time $\widetilde{a_o}$ by

$$\widetilde{a_o} = s_o - \widetilde{j_o} \tag{3.5}$$

where $\widetilde{j_o} = (s_o - \widetilde{\phi_o}) \bmod T_o$ represents the delay such a job may have experienced.

**Example 3.4.** Consider the observer task $\tau_o$ in a task set of 4 periodic tasks (extended from Example 3.3) as shown in the table below.

Table 3.4: An RTS task set of 4 periodic tasks.

|  | $T_i$ | $C_i$ | $\phi_i$ |
|---|---|---|---|
| $\tau_1$ | 15 | 1 | 3 |
| $\tau_o$ | 10 | 4 | 1 |
| $\tau_v$ | 8 | 2 | 2 |
| $\tau_4$ | 6 | 1 | 4 |

Let's assume the system begins at $t = 0$ and the observer task starts collecting its start times for 10 instances from $t = 41$. The collected start times are {41, 53, 61, 71, 81, 92, 101, 111, 121, 133}. By using Equation 3.4, the observer task's phase $\widetilde{\phi_o}$ can be computed by min(41, 43, 41, 41, 41, 42, 41, 41, 41, 43) mod 10 = 1. In this example, $\widetilde{\phi_o} = \phi_o = 1$ and thus the correct observer task's phase is obtained.

### 3.4.4 Reconstructing The Observer Task's Execution Intervals

Based on Theorem 3.4 and Equation 3.5, I developed the following proposition to reconstruct the execution intervals in a period:

**Proposition 3.2.** Assume that, in a DyPS attack, the attacker has reconstructed the observer task's phase as $\widetilde{\phi_o}$. Then, for a given job of the observer task starting at $s_o$, the observer task reconstructs only the execution intervals within $[\widetilde{a_o}, \widetilde{a_o} + T_o - T_v)$ where $\widetilde{a_o}$ is calculated using Equation 3.5.

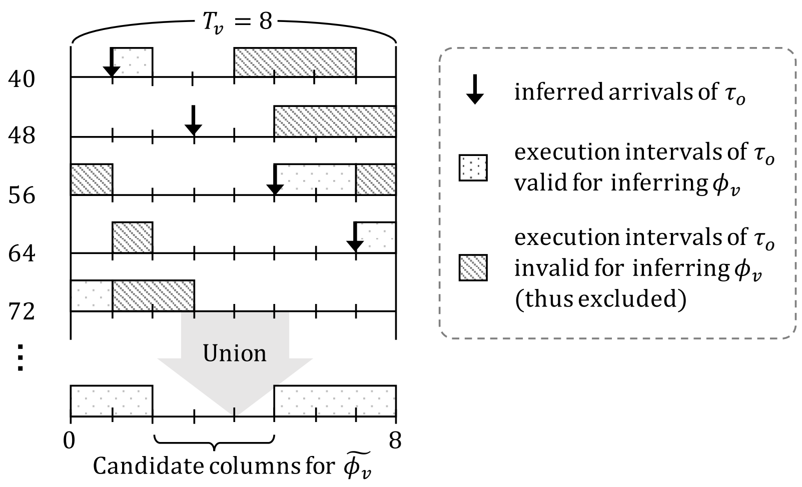
To implement Proposition 3.2, I employ Algorithm 3.1 but make the following modifications: *(i)* at the beginning of the execution of each period, I let the observer task compute $\widetilde{a_o}$ by using Equation 3.5; *(ii)* the observer task stops reconstructing execution intervals in a period if the current time stamp exceeds $\widetilde{a_o} + T_o - T_v$.

The end result of this step is a set of reconstructed execution intervals, denoted by $E_o^{recon} = \{e_o^1, e_o^2, e_o^3, ...\}$ where $e_o^r := [begin_o^r, end_o^r)$ is the $r$-th reconstructed execution interval starting at the time point $begin_o^r$ and ending at the time point $end_o^r$. Note that the number of reconstructed execution intervals are dependent on the duration of the attack that is determined by the attacker. The impact of the attack duration is evaluated in Section 3.7.2.

### 3.4.5 Computing The Candidates

To compute the candidate time points for the phase of the victim task, the reconstructed execution intervals are organized on a timeline with length equal to the victim task's period $T_v$. To facilitate understanding, let us use the *schedule ladder diagram* introduced in Section 3.3.4 (of width $T_v$) to illustrate how the reconstructed execution intervals are processed. On a schedule ladder diagram, the victim task's arrivals are always present in the same column (since the width equals $T_v$.) Let's define such a column as the "true arrival column" that has an offset of $\phi_v$ from the leftmost time column. As the reconstructed execution intervals of $\tau_o$ are ensured to have priorities lower than the victim task, those execution intervals will not appear in the true arrival column (because otherwise the victim task would have executed instead). In other words, the time columns where the reconstructed execution intervals of $\tau_o$ appear even once cannot be the true arrival column.

Given a reconstructed execution interval $e_o^r \in E_o^{recon}$, the time columns where $e_o^r$ is present are determined by $\{t \bmod T_v \mid begin_o^r \leq t < end_o^r \wedge t \in \mathbb{Z}\}$. For simplicity, let's define $e_o^r \bmod T_v := \{t \bmod T_v \mid begin_o^r \leq t < end_o^r \wedge t \in \mathbb{Z}\}$. Therefore, the time columns where the reconstructed execution intervals appear at least once can be calculated



$T_v = 8$
40
48
56
64
72
⋮
Union
0
8
Candidate columns for $\widetilde{\phi_v}$
inferred arrivals of $\tau_o$
execution intervals of $\tau_o$ valid for inferring $\phi_v$
execution intervals of $\tau_o$ invalid for inferring $\phi_v$ (thus excluded)

Figure 3.7: A schedule ladder diagram demonstrated for Example 3.5. Both valid and invalid execution intervals are plotted in the diagram for reference. In the DyPS algorithms, only the valid execution intervals are taken into account for inferring the victim task's phase (Proposition 3.2.)

by $\bigcup_{e_o^r \in E_o^{recon}} (e_o^r \bmod T_v)$ which represents a set of "false" time columns that do not include the true arrival time column. Thus, the set of candidate time columns can be obtained by

$$\overbrace{\{col \mid 0 \le col < T_v \wedge col \in \mathbb{Z}\}}^{\text{all time columns of } \tau_v} - \underbrace{\bigcup_{e_o^r \in E_o^{recon}} (e_o^r \bmod T_v)}_{\text{a set of ``false'' time columns}} \tag{3.6}$$

### 3.4.6 Inferring The Victim Task's Phase

Next, I take the beginning of the longest contiguous time columns in the candidate list as the inference of the victim task's phase, $\widetilde{\phi_v}$. Then, the future arrival time of the victim task can be calculated by $\widetilde{\phi_v} + k \cdot T_v$ where $k$ is the desired arrival number. Alternatively, given a time point $t$, the subsequent arrival time of the victim task can be predicted by $t + (\widetilde{\phi_v} - t) \bmod T_v$.

**Example 3.5.** Let's consider the task set from Example 3.4. Assume that the observer task has collected its execution intervals for a duration of $LCM(T_o, T_v)$[6] (*i.e.*, 40 time units in this example) since the job starts at $t = 41$. The reconstructed execution intervals are $E_o^{recon} = \{[41, 42), [61, 63), [71, 73)\}$ and they correspond to the time columns $\{1\}$, $\{5, 6\}$ and $\{0, 7\}$, respectively. Figure 3.7 displays the reconstructed execution intervals on a schedule ladder

[6]As we will see in Section 3.7.2, the attack duration is evaluated with using $LCM(T_o, T_v)$ as an unit since the offset between the arrivals of $\tau_o$ and $\tau_v$ resets every $LCM(T_o, T_v)$.

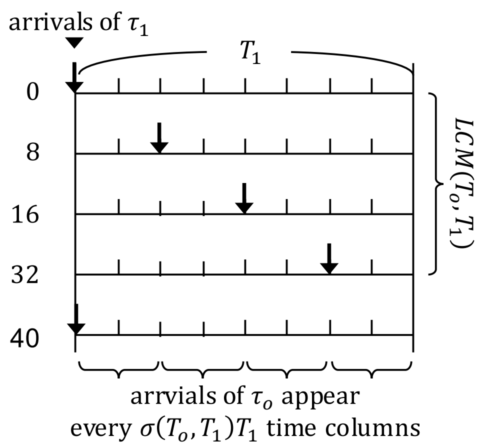

arrivals of $\tau_1$
$T_1$
0
8
16
32
40
$LCM(T_o, T_1)$
arrvials of $\tau_o$ appear
every $\sigma(T_o, T_1)T_1$ time columns

Figure 3.8: An example of the arrival time correlation between $\tau_o$ and $\tau_1$ for Theorem 3.5. In this example, an observer task $\tau_o$ ($T_o = 10$) and a periodic task $\tau_1$ ($T_1 = 8$) are considered. It shows that the projected arrivals of $\tau_o$ appear every $\sigma(T_o, T_1)T_1 = 2$ time columns.

diagram and the timeline at the bottom shows the union of the time columns, $\{0, 1, 5, 6, 7\}$, in which the reconstructed execution intervals appear at least once. The candidate time columns are then computed as $\{0, ..., 7\} - \{0, 1, 5, 6, 7\} = \{2, 3, 4\}$ (Equation 3.6) and the inference is determined as $\widetilde{\phi_v} = 2$ (*i.e.,* the first time column of the longest contiguous time columns, $\{2, 3, 4\}$) which matches the ground truth, $\phi_v = 2$.

It is worth mentioning that the correct $\phi_v$ may not be inferred in Example 3.5 if Proposition 3.2 is not enforced. If all the execution intervals (both valid and invalid execution intervals in Figure 3.7) are considered when calculating the union of the time columns, the true arrival column will be excluded from the candidate time columns.

The aforementioned situation may happen if an incorrect task phase for the observer task is reconstructed in the first place. While it may cause some issues when $\widetilde{\phi_o} \neq \phi_o$, our analysis in Section 3.4.7 shows that it can happen only under certain rare conditions. The experimental results presented in Section 3.7.2 further show that the attacker can get a high inference precision even in those conditions due to the presence of run-time variations.

### 3.4.7 Impact on The Reconstruction of Observer Task's Phase

In Section 3.4.3 I presented algorithms that reconstruct the observer task's phase by using the start times of *its own jobs*. When there exists at least one job of the observer task whose arrival time equals its start time (*i.e.,* the job is not delayed by any higher priority task and thus $a_o = s_o$), the correct task phase can be reconstructed. On the other hand, the correct inference cannot be made if the start of the observer task jobs consistently experience delays in every period. Here I explore and characterize the factors that may contribute to the delays in the observer task's start times by analyzing a simple task set with just two tasks.

**Theorem 3.5.** Given an observer task $\tau_o$ and a periodic task $\tau_i$, the maximum proportion of the arrivals of $\tau_o$ that may experience interference solely due to task $\tau_i$ is computed by

$$\psi(\tau_o, \tau_i) = \left\lceil \frac{\overset{\text{utilization of } \tau_i}{u_i}}{\sigma_{(\tau_o,\tau_i)}} \right\rceil \sigma_{(\tau_o,\tau_i)} \qquad \text{the number of arrivals of } \tau_o \text{ in a } LCM(T_o, T_i) \tag{3.7}$$

where $u_i = \frac{C_i}{T_i}$ is the utilization of $\tau_i$, $\sigma_{(\tau_o,\tau_i)} = \frac{T_o}{LCM(T_o,T_i)}$ is the inverse of the number of arrivals of $\tau_o$ in a $LCM(T_o, T_i)$ and $\psi(\tau_o, \tau_i)$ is a fraction in the range $0 < \psi(\tau_o, \tau_i) \leq 1$.

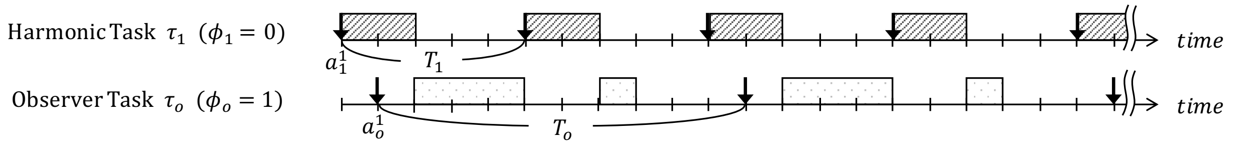
*Proof.* Since both tasks are periodic, the schedule of the two tasks repeats after their *least common multiple*, $LCM(T_o, T_i)$. There are $\frac{LCM(T_o,T_i)}{T_o}$ arrivals of $\tau_o$ in a $LCM(T_o, T_i)$. By projecting these arrivals of $\tau_o$ onto a timeline with length equal to the period of $\tau_i$, an arrival of $\tau_o$ appears every $\frac{1}{\frac{LCM(T_o,T_i)}{T_o}} \cdot T_i = \sigma_{(\tau_o,\tau_i)} T_i$ time units and repeats after each $LCM(T_o, T_i)$ [20, Observation 2], as illustrated in Figure 3.8. The most number of arrivals of $\tau_o$ that may experience interference due to the execution of $\tau_i$ in one $LCM(T_o, T_i)$ is computed by $\left\lceil \frac{C_i}{\sigma_{(\tau_o,\tau_i)} T_i} \right\rceil = \left\lceil \frac{u_i}{\sigma_{(\tau_o,\tau_i)}} \right\rceil$ which happens when there exists an instant at which both tasks arrive at the same time, *e.g.*, when $\phi_o = \phi_i$ or $(\phi_o \bmod T_i) = \phi_i$. In this case, the maximum proportion of the arrivals of $\tau_o$ may be interfered by $\tau_i$ in one $LCM(T_o, T_i)$ can be computed by $\psi(\tau_o, \tau_i) = \left\lceil \frac{u_i}{\sigma_{(\tau_o,\tau_i)}} \right\rceil \sigma_{(\tau_o,\tau_i)}$. Since this relation is the same across all $LCM(T_o, T_i)$, the calculated $\psi(\tau_o, \tau_i)$ applies to the whole schedule. QED.

When $\psi(\tau_o, \tau_i) < 1$, it means that at least $1 - \psi(\tau_o, \tau_i)$ of the arrivals of $\tau_o$ are not interfered by $\tau_i$. In contrast, when $\psi(\tau_o, \tau_i) = 1$, it is possible for the execution of $\tau_i$ to interfere all the arrivals of $\tau_o$, which would result in an inaccurate $\widetilde{\phi_o}$. Intuitively, it can happen when $\tau_i$ has a period in harmony with that of $\tau_o$. An example is given below.

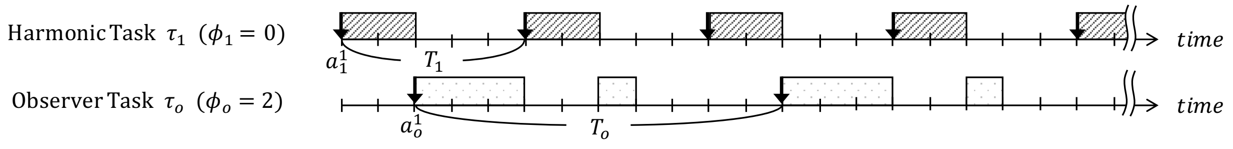
**Example 3.6.** Consider an observer task $\tau_o$ ($T_o = 10$, $C_o = 4$) and a task $\tau_1$ ($T_1 = 5$, $C_1 = 2$). They are in harmony because $T_o \bmod T_1 = 0$. Then $\psi(\tau_o, \tau_1)$ is computed by

$$\sigma_{(\tau_o,\tau_1)} = \frac{T_o}{LCM(T_o, T_1)} = 1 \tag{3.8}$$

$$\psi(\tau_o, \tau_1) = \left\lceil \frac{u_1}{\sigma_{(\tau_o,\tau_1)}} \right\rceil \sigma_{(\tau_o,\tau_1)} = \lceil u_1 \rceil = 1 \tag{3.9}$$


Harmonic Task $\tau_1$ ($\phi_1 = 0$)
Observer Task $\tau_o$ ($\phi_o = 1$)
$a_1^1$
$T_1$
$a_o^1$
$T_o$
time
time

(a) When $\phi_1 = 0$ and $\phi_o = 1$, all arrivals of $\tau_o$ experience interference due to the harmonic task $\tau_1$.


Harmonic Task $\tau_1$ ($\phi_1 = 0$)
Observer Task $\tau_o$ ($\phi_o = 2$)
$a_1^1$
$T_1$
$a_o^1$
$T_o$
time
time

(b) When $\phi_1 = 0$ and $\phi_o = 2$, no arrival of $\tau_o$ experiences interference due to the harmonic task $\tau_1$.

Figure 3.9: Schedules of the task set $\Gamma = \{\tau_o, \tau_1\}$ given in Example 3.6. The $T_o$ and $T_i$ are in harmony and thus $\psi(\tau_o, \tau_1) = 1$.

which indicates that all the arrivals of $\tau_o$ may experience interference due to $\tau_1$. Figure 3.9 illustrates two possible schedules for the given task set. Figure 3.9(a) shows the case where $\phi_o = 1$ and $\phi_1 = 0$, all the arrivals of $\tau_o$ are interfered by $\tau_1$. Figure 3.9(b) shows the case where $\phi_o = 2$ and $\phi_1 = 0$, all the arrivals of $\tau_o$ are no longer interfered by $\tau_1$.

This example shows a crucial fact that $\psi(\tau_o, \tau_i)$ only represents the upper bound of the interference (when only one task is considered). That is, having $\psi(\tau_o, \tau_i) = 1$ does not mean all the arrivals of $\tau_o$ are absolutely interfered. With the same task set but different task phases (which can vary across systems and every time the system restarts), the impact of the interference can be quite different, as shown in Example 3.9.

On the other hand, the schedules presented in Figure 3.9 are generated based on WCETs. However, the actual execution times at run-time in real systems can vary throughout. As a result, the run-time task utilization is in fact smaller and, based on Equation 3.7, the proportion of arrivals of the observer task is also smaller.

Taking the schedule in Figure 3.9(a) as an example, if the first instance's execution time of $\tau_1$ is $c_1^1 = 1$ (rather than its WCET), then the arrival $a_o^1$ of the observer task will not experience any interference. In such a case, $s_o^1 = a_o^1$ and reconstructing the correct task phase becomes possible. Therefore, the actual impact is highly dependent on the task phases as well as the run-time variations.

### 3.4.8 Coverage Ratio in EDF

The coverage ratio $\mathbb{C}(\tau_o, \tau_v) = \frac{C_o}{GCD(T_o, T_v)}$ [20, Definition 1] is used to estimate the proportion of the time columns that can be covered by the execution of the observer task in the FP

RTS. When a given observer task and victim task pair satisfies $\mathbb{C}(\tau_o, \tau_v) \geq 1$, the execution of the observer task may cover all the time columns on the schedule ladder diagram and observing the victim task's arrivals is possible. Since the coverage ratio and the inference precision have a positive correlation, it is useful for evaluating the attacker's capabilities against the victim task.

While the idea of the coverage ratio can be applied in the case of DyPS, the calculation must be revised to reflect the scope of the reconstructed execution intervals specified in Proposition 3.2 before it can be used. Therefore, I redefine the coverage ratio for DyPS as follows:

**Definition 3.3.** ($\mathbb{C}_{DyPS}(\tau_o, \tau_v)$ DyPS Coverage Ratio) The coverage ratio of DyPS in EDF, denoted by $\mathbb{C}_{DyPS}(\tau_o, \tau_v)$, is computed by

$$\mathbb{C}_{DyPS}(\tau_o, \tau_v) = \frac{\overbrace{\min(C_o, T_o - T_v)}^{\text{viable observation length}}}{\underbrace{GCD(T_o, T_v)}_{\text{greatest common divisor of } T_o \text{ and } T_v}} \tag{3.10}$$

It represents the proportion of the time columns where the observer task's (valid) execution can potentially be present in the schedule ladder diagram. If all $T_v$ time columns can be covered by the observer task, then $\mathbb{C}_{DyPS}(\tau_o, \tau_v) \geq 1$. Otherwise $0 < \mathbb{C}_{DyPS}(\tau_o, \tau_v) < 1$.

## 3.5 IMPLEMENTATION IN REAL-TIME LINUX

I implemented both the ScheduLeak algorithms and the Dyps algorithms in both a Linux-based platform and a simulator. In this section I introduce the implementation in Linux. Design space exploration through simulations is presented in Section 3.7.

### 3.5.1 Platform and Operating System

I used a *Raspberry Pi 3 Model B* (RPi3B) development board as the base platform to implement the ScheduLeak and the DyPS algorithms. RPi3B is equipped with a 1.2 GHz ARM Cortex-A53 CPU and runs on a vendor-supported open-source operating system, Raspbian (a variant of Debian Linux). Specifically, I used a real-time Linux kernel (*i.e.,* a Linux kernel with a `PREEMPT_RT` patch applied) and configured the system correspondingly as detailed in Table 3.5 for satisfying the introduced real-time system model. To run real-time tasks, I

Table 3.5: Summary of the Implementation Platform

| **Artifact** | **Parameters** |
|---|---|
| Platform | Raspberry Pi 3 Model B (RPi3B) Development Board |
| Specifications | ARM Cortex-A53, 1.2 GHz 64-bit processor, 1 GB RAM |
| Operating System | Debian Linux (Raspbian) Kernel 4.19.59-rt24 |
| Kernel Configuration | `CONFIG_PREEMPT_RT_FULL` enabled |
| Boot Commands | `maxcpus`=1 |
| Run-time Variables | `sched_rt_runtime_us`=−1, `scaling_governor`=`performance` |
| Scheduler | `SCHED_DEADLINE` |

used the built-in real-time scheduler, `SCHED_DEADLINE` [65] that has been maintained in the mainline Linux kernel since version 3.14 as an EDF scheduler and `SCHED_FIFO` as an RM scheduler.

### 3.5.2 Implementation Details

My implementation of the attack algorithms is wrapped in a periodic real-time task acting as an observer task. For the purpose of testing, the observer task is made configurable by accepting arguments passed from the command line interface upon creation. This allows me to easily create an observer task with desired period, WCET, ladder diagram width and attack duration in experiments.

In the implementation of DyPS, I combined the first two steps (Section 3.4.3 and 3.4.4) by making use of the same set of execution intervals for inferring the observer task's phase and extracting the valid part of the execution intervals. To do so, I first let the observer task collect its start times and reconstruct the whole part of the corresponding execution intervals (which would include both valid and invalid parts) for a given attack duration. Then, the start times are used to infer the observer task's phase which is further used to exclude the invalid part of the reconstructed execution intervals. The resulting, valid execution intervals are consumed for producing an inference of the victim task's phase, as introduced in Section 3.4.5 and 3.4.6. Note that, while most of the algorithm implementation involves mathematical computation, the implementation for reconstructing the execution intervals is hardware-dependent. I now elaborate on the latter part using RPi3B as an example.

For an observer task to reconstruct its own execution intervals it needs the ability to monitor any preemptions that occur during its execution. In a Linux system, this can be done by letting the observer task consistently read time counts from the `CLOCK_MONOTONIC` clock source by invoking the unprivileged system call `clock_gettime`() that yields counts in nanoseconds. When no preemption has occurred, the difference between two adjacent time counts should contain only the minimum computation overhead (*i.e.,* a time reading call and a condition check for the loop that encapsulates this procedure.) On RPi3B, this overhead is 441.56$ns$ (averaged from 1000 samples.) This measurement can be used as a reference to determine if a preemption has occurred. In my implementation, I take a tenfold threshold (5000$ns$) to leave sufficient room for variations. When the difference between two adjacent time counts is greater than 5000$ns$, the execution is deemed being preempted which signals the end of the previous execution interval and the start of a new execution interval.

## 3.6 EVALUATION METRICS AND SETUP

### 3.6.1 Evaluation Metrics

There are mainly two attack stages in the DyPS algorithms: *(i)* reconstructing $\phi_o$ for determining valid execution intervals and *(ii)* inferring $\phi_v$ for predicting future arrival time points. While both stages target the computing of a task's phase, they have very different characteristics due to how they are inferred. I use two different metrics to evaluate the results from the two stages as defined next.

**Reconstructing The Observer Task's Phase.** As introduced in Section 3.4.3, the observer task's phase is reconstructed based on the collected start times where $s_o \geq a_o$ (*i.e.,* the start times are always on the right of the corresponding arrival times), thus the distance between $\widetilde{\phi_o}$ and $\phi_o$, denoted by $\Delta\widetilde{\phi_o} = (\widetilde{\phi_o} - \phi_o) \bmod T_o$, should always be positive. Based on this fact, I define the error ratio for $\widetilde{\phi_o}$ as follows:

**Definition 3.4.** ($\mathbb{E}^o$ Error Ratio) The error ratio of $\widetilde{\phi_o}$, denoted by $\mathbb{E}^o$, is computed by

$$\mathbb{E}^o = \frac{\overbrace{\Delta\widetilde{\phi_o}}^{\text{distance between } \phi_o \text{ and a projected } \widetilde{\phi_o} \text{ on its right}}}{\underbrace{T_o}_{\text{the observer task's period}}} \tag{3.11}$$

where $\Delta\widetilde{\phi_o} = (\widetilde{\phi_o} - \phi_o) \bmod T_o$ represents the distance between $\phi_o$ and a projected $\widetilde{\phi_o}$ on its right. The resulting $\mathbb{E}^o$ value is a real number in the range $0 \leq \mathbb{E}^o < 1$. A smaller $\mathbb{E}^o$ means that the reconstructed $\widetilde{\phi_o}$ has less error when compared to the true $\phi_o$.

Note that $\mathbb{E}^o$ is bounded by 1 because the start times used for computing $\widetilde{\phi_o}$ are bounded by $a_o \leq s_o < a_o + T_o$.

**Inferring The Victim Task's Phase.** The second stage of the attack is to infer the task phase of the victim task which is the same as the main stage in the ScheduLeak attack in the FP RTS. In contrast to the observer task's phase, I'm only concerned with how close the inference $\widetilde{\phi_v}$ is to the actual $\phi_v$ and $\widetilde{\phi_v}$ can be on either side of $\phi_v$. Here, I define inference precision $\mathbb{I}_v^o$ as follows:

**Definition 3.5.** ($\mathbb{I}_v^o$ Inference Precision of $\widetilde{\phi_v}$) The inference precision, denoted by $\mathbb{I}_v^o$, is computed by

$$\mathbb{I}_v^o = \left| \overset{\text{the smallest distance between the inferred and the true task phases}}{\frac{\epsilon}{T_v/2}} - 1 \right| \tag{3.12}$$

where $\epsilon = \left|\widetilde{\phi_v} - \phi_v\right|$. The resulting $\mathbb{I}_v^o$ value is a real number in the range $0 < \mathbb{I}_v^o \leq 1$. A larger $\mathbb{I}_v^o$ indicates that the inference $\widetilde{\phi_v}$ is more precise in inferring $\phi_v$.

Based on the inference precision, I further define an inference to be successful if the attacker is able to *exactly* infer the victim task's phase. Therefore, the result of an inference is either true or false. The *inference success rate* is then defined as the percentage of the tested task sets in which the inferences are successful for a given test condition.

### 3.6.2 Experiment Setup

I test the introduced attack algorithms with randomly generated synthetic task sets. The task sets are grouped by CPU utilization from $[0.001 + 0.1 \cdot x, 0.1 + 0.1 \cdot x]$ where $0 \leq x \leq 9$. Each utilization group consists of 6 subgroups that have a fixed number of tasks $(5, 7, 9, 11, 13, 15)$. Each subgroup contains 100 task sets. In each task set, 50% of the tasks are generated as periodic tasks $(3, 4, 5, 6, 7, 8$ periodic tasks for each subgroup respectively) while the rest of the tasks are generated as sporadic tasks. The task periods are randomly drawn from $[100, 1000]$ and I assume that the attacker has access to the system time with a resolution of 1. The task initial offset is randomly selected from $[0, T_i)$. In the case of sporadic tasks, I take the generated task period as the minimum inter-arrival time. In the case of ScheduLeak against FP RTS, the task priorities are assigned using the rate-monotonic algorithm [9]. I only pick those task sets that are schedulable.

The observer task and the victim task are assigned when generating the task sets. In simulations, I consider a periodic observer task because it represents the worst case attack scenario for the adversary, as discussed in Section 3.3.6. In the case of ScheduLeak in FP RTS, since only the tasks with higher priorities influence the observations, I skip the generation of lower-priority tasks $lp(\tau_o)$. Thus, the observer task always has the lowest priority (*i.e., $pri_o = 1$*) in these generated task sets. For the victim task, two conditions are considered: *(i) $pri_v = 2$* and *(ii) $pri_v = |hp(\tau_o)|$*. This is to test the two boundary conditions. Further, I set the coverage ratio to be $\mathbb{C}(\tau_o, \tau_v) \geq 1$ when generating the task sets (except for evaluating the impact of the coverage ratio), to evaluate whether the algorithms can truly produce confident inferences while the attacker has theoretical guarantees of the attack capability (*i.e.,* having full coverage of all $T_v$ time columns, as per Theorem 3.2). The maximum construction duration $\lambda$ is set as per Section 3.3.7. Thus, $\lambda = GCD(T_o, T_v)$.

In the case of DyPS in DP RTS, the observer task and the victim task in a task set are selected from the generated tasks based on the task periods. 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 task is then selected as 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 $T_o > T_v$ (an assumption from Theorem 3.3) and that there exist other tasks with diverse periods (*i.e.,* some with smaller periods and some with larger periods compared to $T_o$ and $T_v$.) The task sets are generated with $\mathbb{C}_{ScheduLeak} \geq 1$ (except for the experiment that evaluates the impact of the ScheduLeak coverage ratio). It is to examine the performance of the ScheduLeak algorithms under the best case (*i.e.,* all $T_v$ time columns on the schedule ladder diagram may be covered by the observer task's execution.)

For varying the execution times of the tasks and adding jitter to the inter-arrival times (for the sporadic tasks), I use the normal and Poisson distributions respectively. Note that Poisson distribution is used for inter-arrival time variation because the probability of each occurrence (*i.e.,* each arrival of the sporadic task) is independent in such a distribution model.

First, a schedulable task set is generated (using the aforementioned parameters). Then, for a task $\tau_i$, the average execution time is computed by $wcet_i \cdot 80\%$. Next, I fit a normal distribution $\mathcal{N}(\mu, \sigma^2)$ for the task $\tau_i$. I let the mean value $\mu$ be $wcet_i \cdot 80\%$ and find the standard deviation $\sigma$ with which the cumulative probability $P(X \leq wcet_i)$ is 99.99%. As a result, such a normal distribution produces variation such that 95% of the execution times are within $\pm 10\% \cdot wcet_i$. To ensure that the task set remains schedulable, I adjust the maximum modified execution time to be equal to WCET if it exceeds WCET. For sporadic tasks, the average inter-arrival time is computed by $T_i \cdot 120\%$. I use a Poisson distribution with $T_i \cdot 120\%$ as its mean value to generate the varied inter-arrival times during the simulation. Similarly, so as to not violate the given minimum inter-arrival time for a sporadic task, I regenerate the modified inter-arrival time if it drops below $T_i$.

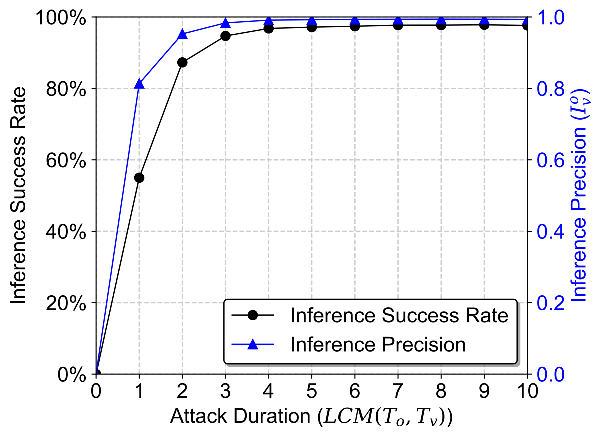


Figure 3.10: The results of the success rate and the inference precision ratio by running the algorithms for different lengths of time. It indicates that longer attack durations can increase the chance of success and yield better inference precision. The points are connected only as a guide.

## 3.7 EVALUATION RESULTS

### 3.7.1 The ScheduLeak Attack

**Attack Duration.** Our first goal is to understand the effects of how long attacks last. Recall that the coverage of the schedule ladder diagram repeats every $LCM(T_o, T_v)$ (Observation 3.2). Therefore, I use $LCM(T_o, T_v)$ as the unit of time to evaluate the algorithms. Taking the Ardupilot software as an example, the largest $LCM$ of any real-time task (*i.e.,* a AP_HAL thread) pairs is $20ms$. While $LCM(T_o, T_v)$ varies system to system, this gives us an insight into the scale of $LCM(T_o, T_v)$. In this experiment, I generate task sets as explained in Section 3.6.2 and run the ScheduLeak algorithms with a fixed duration of $10 \cdot LCM(T_o, T_v)$ for every task set. Figure 3.10 shows the results of this experiment. In Figure 3.10, each point of the inference precision ratio is the mean of the individual inference precision ratios of 12000 task sets for a given attack duration. The results suggest that the longer the attack is sustained, the higher success rate and precision ratio the algorithms can achieve. This is because a longer attack time means more execution intervals are reconstructed by the observer task. On the other hand, both success rate and precision ratio plateau after $5 \cdot LCM(T_o, T_v)$ with the success rate and the precision ratio higher than 97% and 0.99 respectively. This shows that the proposed algorithms can produce inference with precision in a very short time and the additional gains obtained from running longer are minuscule. For this reason, *I evaluate the algorithms with a duration of* $10 \cdot LCM(T_o, T_v)$ *for the rest of the experiments below.*

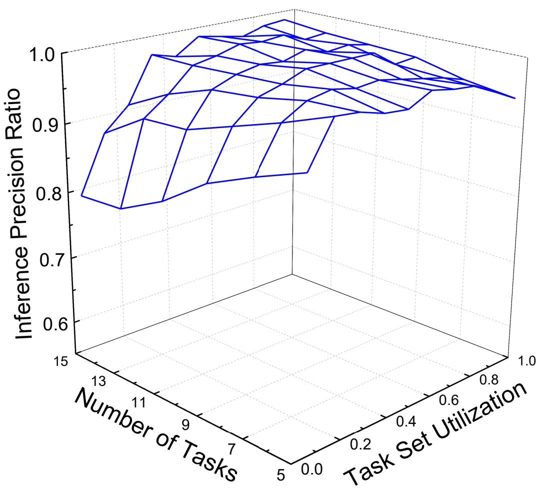
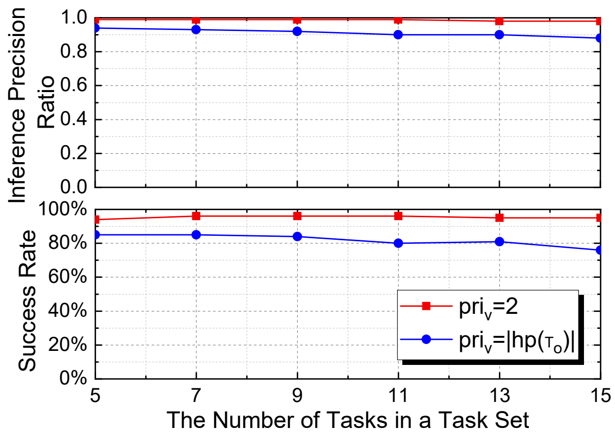


Figure 3.11: The impact of the number of tasks in a task set and the task set utilization. The result shows that the algorithms perform better with small number of tasks and high task set utilization.

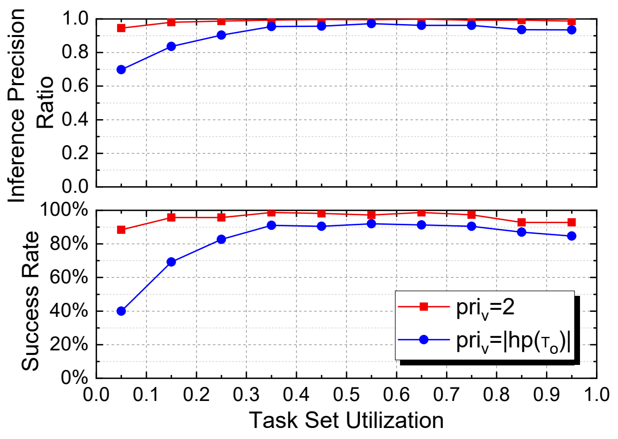
**The Number of Tasks and Task Set Utilization.** Figure 3.11 displays a 3D graph that shows the averaged inference precision ratio for each combination of the number of tasks and the task utilization subgroup. The results suggest that *(i)* the inference precision ratio decreases as the number of tasks in a task set increases and *(ii)* the inference precision ratio increases as the task set utilization increases. The worst inference precision ratio happens when there are 15 tasks in a task set with the utilization group $[0.001, 0.1]$ – these are boundary conditions for both the number tasks and the utilization in this experiment.

The impact of the number of tasks is straightforward as having more tasks in $hp(\tau_o)$ means that $\tau_o$ will be preempted more frequently. This makes it hard for the observer task to eliminate the false time columns. For the impact of the task set utilization, a low utilization value implies that the execution times of the tasks are small and there exists a lot of gaps in the schedule. Hence, the observer may get many small and scattered intervals. Since I let the algorithms pick the largest interval to infer the true arrival column, multiple small intervals are problematic – the algorithm has a hard time picking the right interval that contains the true arrival. Hence errors are compounded.

**Priority of the Victim Task.** Here I analyze the impact of the victim task's priority in a task set. From Section 3.6.2, I consider two boundary conditions for the victim task's position: *(i)* $pri_v = 2$ and *(ii)* $pri_v = |hp(\tau_o)|$. Figures 3.12(a) and (b) present the experiment results for the two conditions. Figure 3.12(a) shows that the huge drop in Figure 3.11 (as the number of tasks increases) is mainly caused by the condition $pri_v = |hp(\tau_o)|$. Figure 3.12(b)



(a) Grouped by the number of tasks.



(b) Grouped by task set utilization.

Figure 3.12: The impact of the victim task's position in a task set. It suggests that a victim task with higher priority makes it hard for the algorithms to make a correct inference. This result stands throughout different number of tasks in a task set as well as different task set utilization. Also, a high priority victim task with low task set utilization reduces the inference performance. This explains the huge drop in Figure 3.11.

also shows the similar indication that the drop in low utilization groups in Figure 3.11 is a result of the condition $pri_v = |hp(\tau_o)|$. It's worth noting that, since I use the rate-monotonic algorithm to assign the priority, $pri_v = 2$ means that $\tau_v$ has a large period, hence potentially has greater execution time. It benefits the algorithms as I pick the largest interval to make an inference in the final step.

**Sporadic and Periodic Tasks.** I examine the impact of the mix of sporadic and periodic tasks. I generate task sets with 0%, 25%, 50%, 75% and 100% sporadic tasks in a task set. The rest of the tasks in a task set are periodic tasks. Comparing the result of all periodic tasks and the result of all sporadic tasks shown in Figure 3.13, I find that the *algorithms perform better with more sporadic tasks*. It shows an ascending trend as the proportion of sporadic tasks increases. However, the change in the performance is less than 1%, which is subtle. Hence, our inference algorithms are fairly agnostic to the actual mix of sporadic/periodic tasks in the system.

**Coverage Ratio and The Maximum Reconstruction Duration.** The experiments above show that the algorithms can reach certain inference success rates and precision when $\mathbb{C}(\tau_o, \tau_v) \geq 1$ and $\lambda = GCD(T_o, T_v)$. However, attackers may face a victim system where $\mathbb{C}(\tau_o, \tau_v) < 1$. That is, the observer task's execution is not guaranteed to appear in all $T_v$ time columns. To evaluate the performance of the algorithms against such a case, I

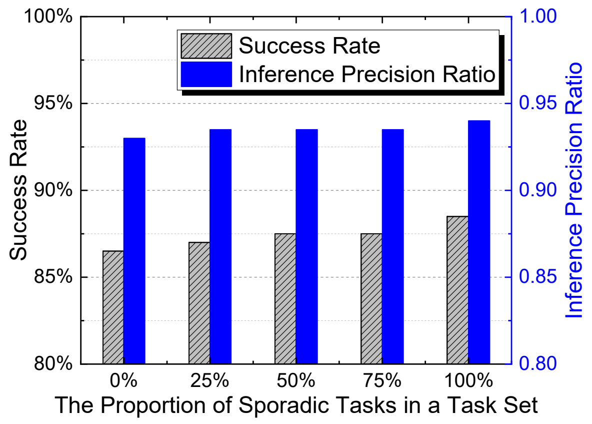


Figure 3.13: The impact of sporadic tasks and periodic tasks. It indicates that the algorithms perform better with sporadic tasks, with a (slightly) ascending trend as the proportion of sporadic tasks increases.

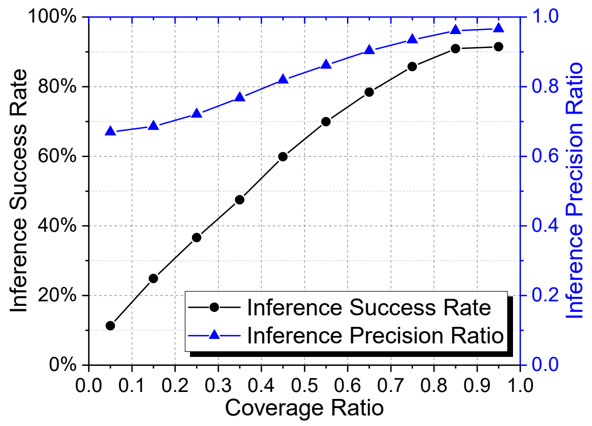


Figure 3.14: The performance of the algorithms when the coverage ratio is less than one.

generate task sets with $0 < \mathbb{C}(\tau_o, \tau_v) < 1$ (thus $\lambda = C_o$) and run the algorithms for a duration of $10 \cdot LCM(T_o, T_v)$. In this experiment, task sets are grouped by coverage ratio from $[0.001 + 0.1 \cdot x, 0.1 + 0.1 \cdot x]$ where $0 \leq x \leq 9$. Figure 3.14 shows the results. It suggests that the attacker may fail to completely infer the victim task's initial offset when the coverage ratio is low. Yet, the algorithms can still succeed in some cases due to the fact that Theorem 3.1 holds even with a low coverage ratio. When the observer has about half coverage of the time columns (the group of $[0.401, 0.5]$), it yields 59.9% in success rate and 0.819 for the averaged inference precision ratio. As more time columns are observed by the observer task, the precision and success rate increase. This is because higher coverage ratios give the algorithms a higher chance to capture the true arrival column and remove others. As a result, the inference success rate is about proportional to the coverage ratio.

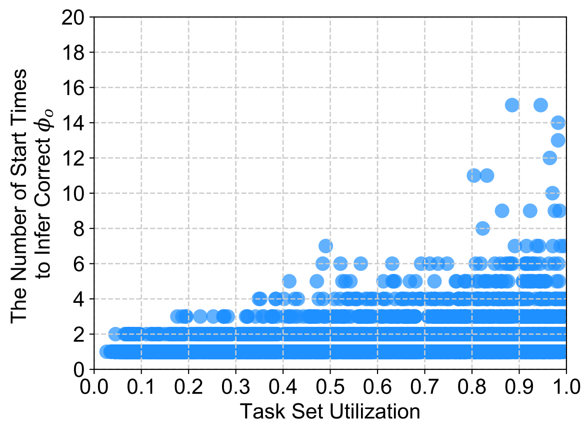


Figure 3.15: The number of start times the DyPS algorithms needs for processing each of the 6000 task sets (without harmonic tasks) to get the observer task's correct phase. Each dot in the figure represents the result of a task set. It shows that more start times are needed when the task set utilization is higher. Yet, the worst case (*i.e.,* 15 start times which corresponds to $15 \cdot T_o$ attack duration) is reasonably small and manageable.

### 3.7.2 The DyPS Attack

**Reconstructing The Observer Task's Phase.** As introduced in Section 3.4.3, the observer task's phase has to be reconstructed before inferring the victim task's phase. Therefore, I evaluate the factors that impact the reconstruction of the observer task's phase.

I first test the DyPS algorithms without any tasks that are in harmony with the observer task. The result shows that the correct $\phi_o$ (*i.e.,* $\widetilde{\phi_o} = \phi_o$ or $\mathbb{E}^o = 0$) can be reconstructed in all of the tested 6000 task sets (without harmonic tasks). The number of start times that the DyPS algorithms process to get $\mathbb{E}^o = 0$ is plotted in Figure 3.15. The figure shows a trend that it requires more start times for the DyPS algorithms to reconstruct correct $\phi_o$ when the utilization is higher. This is because a higher utilization implies higher chances of preemptions and delays in task executions. Nevertheless, even in the worst case, of 6000 tested task sets, a reasonably small number of start times were required (*i.e.,* 15 start times). This shows that the DyPS algorithms are practical for reconstructing $\phi_o$.

Next I evaluate the impact of harmonic tasks. As pointed out in Section 3.4.7, a task with a period that is in harmony with the period of the observer task can contribute a constant delay to the observer task's start times. Therefore, I regenerate 6000 task sets with at least one task in harmony with the observer task in each task set and test with the DyPS algorithms. In this experiment, I let DyPS collect start times within a duration of $10 \cdot LCM(T_o, T_v)$. Each task set is tested with two conditions: *(i)* without run-time variations and *(ii)* with run-time variations. It is to examine the impact of the constant delay caused by the harmonic tasks in both a static and a more realistic RTS environment. Without run-time variations, the

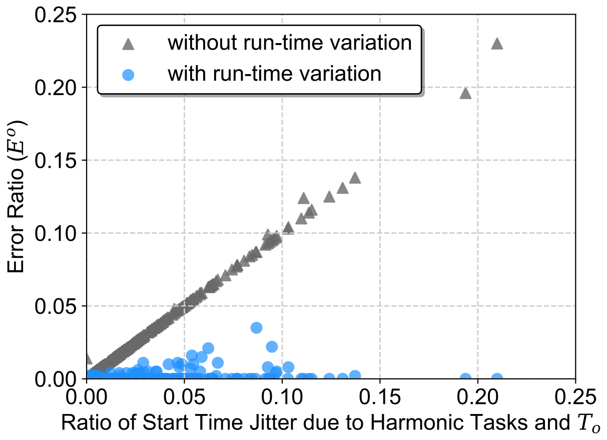


Figure 3.16: The impact of the start time delays caused by the tasks in harmony with the observer task on error ratio. The tested 6000 task sets are generated with at least one task in harmony with the observer task in each task set. Each task set is tested with run-time variations and without run-time variations. The results show that the impact of the start time delays becomes subtle in the presence of run-time variations.

task's execution times will run up to the WCET in every job instance which should retain the constant delay contributed by the harmonic tasks (or any potential delay contributed by other tasks). In contrast, with run-time variations, the task's execution times include variations that are drawn from a normal distribution that gives a more realistic run-time result. The results are plotted in Figure 3.16 and show that, without run-time variations, the error ratio $\mathbb{E}^o$ is proportional to the ratio of the constant delay caused by the harmonic tasks and the observer task's period $T_o$. The outliers for the triangular points are the cases when there are other tasks contributing to the start time delays within the tested attack duration (*i.e.,* $10 \cdot LCM(T_o, T_v)$). It is worth noting that only 6.75% of the task sets have constant delay and $\mathbb{E}^o > 0$ since the task phases are randomly generated and hence it is not guaranteed that the harmonic tasks can always interfere with the observer task's arrivals. This also indicates that having harmonic tasks does not necessarily downgrade the proposed attack. On the other hand, with execution time variations, the impact of the constant start time delay is significantly reduced due to the varied execution times. The DyPS algorithms yield better error ratios in all of the 6000 task sets (88.4% of the task sets that have $\mathbb{E}^o > 0$ without run-time variations yield $\mathbb{E}^o = 0$ with run-time variations.)

To understand the impact of the error ratio of $\widetilde{\phi_o}$ on the inference precision of $\widetilde{\phi_v}$, I test the aforementioned 6000 task sets with synthetically generated $\widetilde{\phi_o}$ from $\{\phi_o + (0.01x)T_o \mid 0 \leq x \leq 10 \wedge x \in \mathbb{Z}\}$ which is expected to yield error ratio in $\{0, 0.01, ..., 0.1\}$. This range is chosen based on the experiment results shown in Figure 3.16 where overall error ratio is smaller than 0.034 with run-time variations. The experiment is carried out with a duration

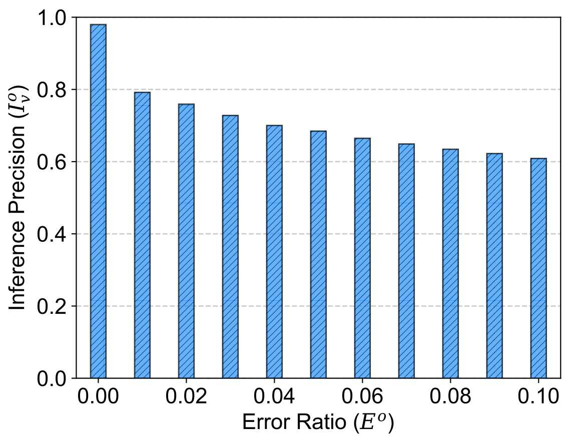


Figure 3.17: The impact of the error ratio on the inference precision. Each bin is the result of the 6000 task sets (with harmonic tasks) with synthetically generated $\widetilde{\phi_o}$ from $\{\phi_o + (0.01x)T_o \mid 0 \leq x \leq 10 \wedge x \in \mathbb{Z}\}$. It shows that the error ratio has negative impact on the inference precision.

of $10 \cdot LCM(T_o, T_v)$ and with run-time variations enabled. Results shown in Figure 3.17 indicate that the error ratio has considerable negative impact on the inference precision. For example, inference precision $\mathbb{I}_v^0$ reduces to 0.79 when $\mathbb{E}^o = 0.01$ from $\mathbb{I}_v^0 = 0.98$ when $\mathbb{E}^o = 0$. Nevertheless, considering only 0.7% of the task sets have $\mathbb{E}^o > 0$ due to the run-time variations and the varied task phases, a high $\mathbb{E}^o$ is arguably uncommon in real cases.

**Inferring The Victim Task's Phase.** I now focus on evaluating the inference precision of $\widetilde{\phi_v}$. Note that the complete DyPS algorithms (including inferring $\widetilde{\phi_o}$) are tested in the experiments here. I first examine the impact of the attack duration on the attack results. I use $LCM(T_o, T_v)$ as an unit of the attack duration to evaluate DyPS since the offset between the observer task and the victim task repeats every $LCM(T_o, T_v)$. Task sets are tested with the attack duration varying from $LCM(T_o, T_v)$ to $10 \cdot LCM(T_o, T_v)$ and the results are plotted in Figure 3.18. As shown, a longer attack duration leads to a higher inference precision. It is because more execution intervals are reconstructed as the attack lasts longer. The inference precision reaches $\mathbb{I}_v^o = 0.978$ at $10 \cdot LCM(T_o, T_v)$ and plateaus afterward. Therefore, I choose $10 \cdot LCM(T_o, T_v)$ as an attack duration for the experiments presented in this section unless otherwise stated.

Next I break down the number of tasks in a task set and the task utilization to evaluate their impact on the inference precision. The experiment results are plotted in Figure 3.19. Each grid in the figure shows the mean inference precision of 100 task sets with the corresponding number of tasks in a task set and the task utilization. A brighter grid has a lower inference precision while a darker grid indicates that the attack yields a better inference

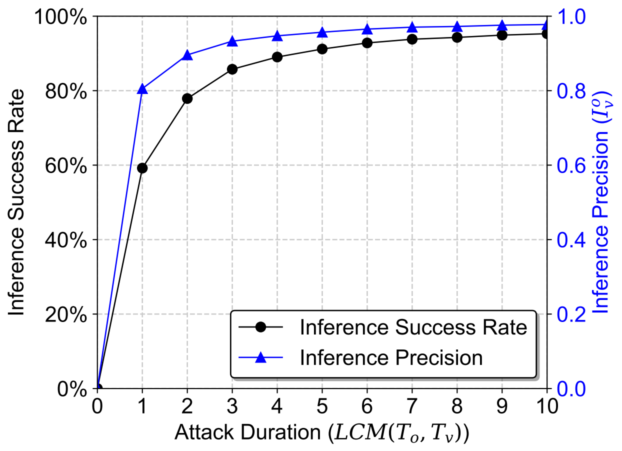


Figure 3.18: The inference precision with varying attack duration. It shows that the longer the attack persists the higher inference precision DyPS can achieve. The inference precision reaches $\mathbb{I}_v^o = 0.978$ at $10 \cdot LCM(T_o, T_v)$ and plateaus afterward.

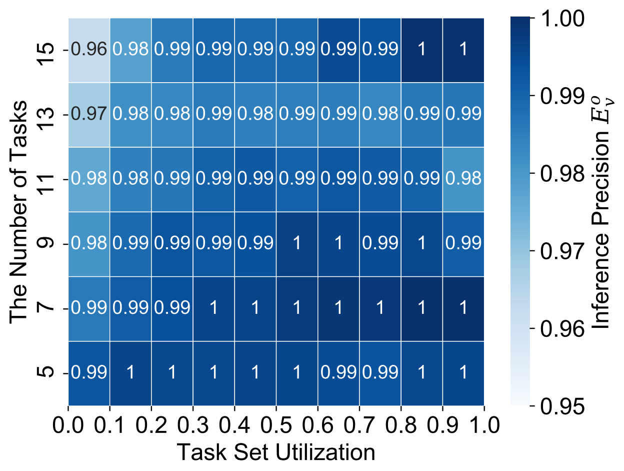


Figure 3.19: The impact of the number of tasks in a task set and the task set utilization on the inference precision. Each grid displays the mean inference precision for the corresponding number of tasks (Y-axis) and task utilization (X-axis). A darker grid shows a higher inference precision.

precision. The resulting heat map gives an intuition for the distribution of the inference precision with varying number of tasks and the task utilization. The figure shows a small degradation when the number of tasks in a task set is high and the utilization is low. It is because a task set with a higher number of tasks can have more tasks preempting and delaying the observer task's execution and this introduces more perturbations to the algorithms. On the other hand, a low utilization implies a low execution time for the observer task which results in shorter execution intervals to be reconstructed. This fact makes it difficult to effectively eliminate false time columns and leads to more scattered candidate time slots for the last step of the algorithms, which causes more uncertainty to the inferences.

**Impact of DyPS Coverage Ratio.** The DyPS coverage ratio $\mathbb{C}_{DyPS}$ represents the proportion of the time columns in a schedule ladder diagram that can be covered by the observer task's execution. In the previous experiments with $\mathbb{C}_{DyPS} \geq 1$, I showed that the DyPS algorithms are able to yield competitive inference precisions in various task set conditions. Here, I evaluate the case when the DyPS coverage ratio is less than one (*i.e.,* $0 < \mathbb{C}_{DyPS} < 1$.) In this experiment, I generate 6000 task sets for each of the DyPS coverage ratio groups from $\{[0.001 + 0.1 \cdot x, 0.1 + 0.1 \cdot x) \mid 0 \leq x \leq 9 \wedge x \in \mathbb{Z}\}$. The mean inference precision is then taken from each DyPS coverage ratio group and the results are plotted in Figure 3.20. It shows that the DyPS algorithms get worse results as the DyPS coverage ratio

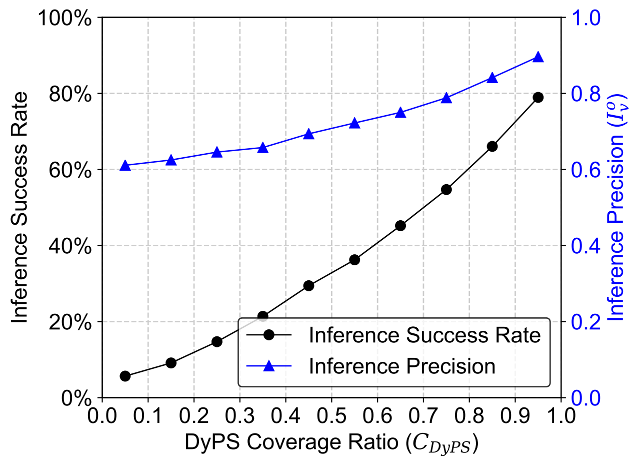


Figure 3.20: The inference precision with varying the DyPS coverage ratio in the range of $0 < \mathbb{C}_{DyPS} < 1$. It suggests that the DyPS algorithms get better performance as the DyPS coverage ratio increases. The DyPS algorithms perform better than a random guess when $\mathbb{C}_{DyPS} \approx 0$.

drops. When $\mathbb{C}_{DyPS} \approx 0$, the DyPS algorithms yield a inference precision ($\mathbb{I}_v^o = 0.61$) that is close to, but still better than, a random guess. Conversely, a higher DyPS coverage ratio has better inference precision since the observer task can cover more time columns on the schedule ladder diagram and hence has a higher chance to encapsulate the true time column.

**Comparison with ScheduLeak.** The main challenge in attacking the EDF RTS is the existence of the invalid part of the execution intervals. To understand the impact on the ScheduLeak algorithms and how the DyPS algorithms handle such intervals, I test both algorithms in the EDF RTS with 6000 newly generated task sets (with $C_o > (T_o - T_v)$, task utilization drawn from $[0.001, 1.0)$) in which the presence of the invalid execution intervals is guaranteed. Results presented in Figure 3.21 show that the DyPS algorithms outperform the ScheduLeak algorithms in handling the invalid execution intervals. As the attack lasts longer, more invalid execution intervals are collected in the case of ScheduLeak that leads to worse inference precision. The ScheduLeak algorithms yield a mean inference prevision of 0.54 at $10 \cdot LCM(T_o, T_v)$. This is only sightly better than a naive attack with random guesses indicating ScheduLeak is inadequate for attacking the EDF RTS.

**Attack Overhead in Linux.** As introduced earlier, my implementation of DyPS mainly has two execution phases: reconstruction and inferences. Here I'm interested in learning the attack overhead for the inferences. To measure the actual overhead on RPi3B, I test my DyPS implementation against 6000 real-time task sets with synthetic task loads. To generate the task parameters for these task sets, I employ the task set generator used in the

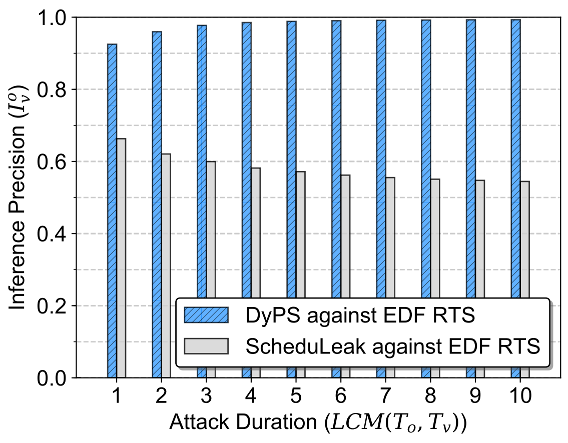
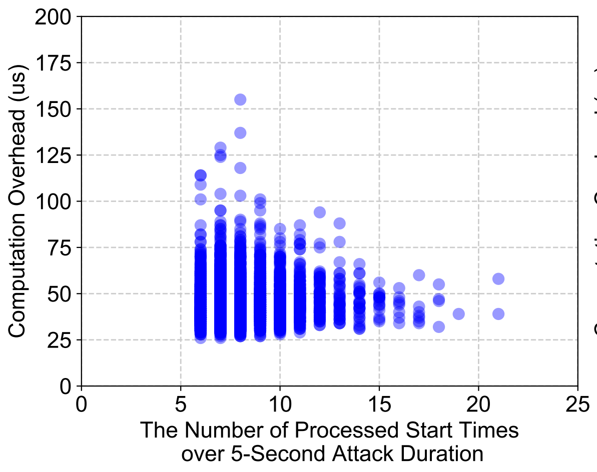


Figure 3.21: The results of employing the state-of-the-art (ScheduLeak) and the proposed ScheduLeak algorithms in the scheduler side-channel attack against the EDF RTS. The end (mean) inference precision of ScheduLeak is 0.54 that is only slightly better than a naive attack with random guesses.

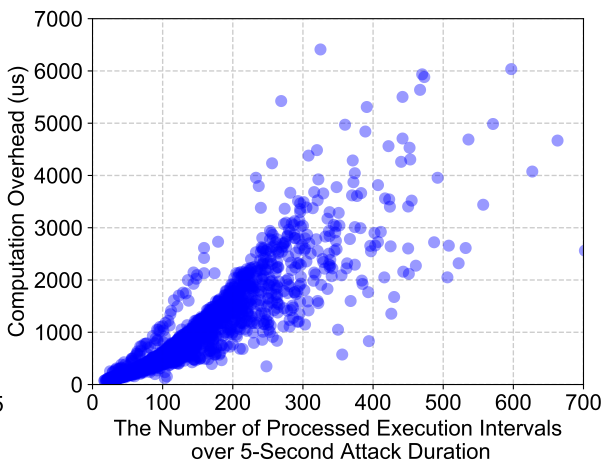
simulation and all generated values are taken as milliseconds. For each test against a task set, I let the observer task observe (*i.e.,* reconstruct execution intervals) for 5 seconds and then measure its computation overhead while producing its inferences.

I first focus on inferring the observer task's phase. Here, the observer task iterates through each of the collected start times to compute the inference. This process has a linear time complexity that is proportional to the number of collected start times. Figure 3.22(a) shows the actual overhead measurement for the 6000 task sets tested on the RPi3B. As each iteration mainly consists of a simple integer comparison, the actual overhead is small (all overhead measurements are smaller than $155us$ in this experiment) and thus the linearity is not noticeable from the resulting plot.

Next I focus on inferring the victim task's phase. In this step, the observer task first excludes the invalid part of the execution intervals, computes a candidate list and then yields an inference. This process has a time complexity of $O(nm)$ where $n$ is the number of reconstructed execution intervals and $m$ is the number of disjoint intervals that exist in the timeline for computing the inference candidates. The actual overhead measurement is shown in Figure 3.22(b). The resulting data points display a trend that matches the aforementioned complexity. As more execution intervals are reconstructed and processed, the computation overhead increases. It is worth noting that 99.6% of the resulting computation overhead is below $4000us$ in a 5-second duration. Considering that the task parameters are on the order of 100 to 1000 milliseconds, inference overheads of $5ms$ or less seem practical. Further, a smarter attack implementation can spread the inference computation across many observer task periods so the execution can stay well within its WCET.



(a) Inferring the observer task's phase.



(b) Inferring the victim task's phase.

Figure 3.22: The computation overhead measured on the RPi3B platform. Each data point in the figures represents the result from a task set and there are 6000 task sets tested. The X-axis in (a) is the number of collected start times and the x-axis in (b) is the number of reconstructed execution intervals over a 5-second duration. The y-axis in both figures is the corresponding computation overhead for processing the collected/reconstructed information.

## 3.8 CASE STUDIES ON REAL PLATFORMS

In what follows, two attack cases based on ScheduLeak in FP RTS are presented. They benefit from the information obtained by the proposed algorithms and utilize such information to accomplish their primary attack goals. The demo videos for these attack cases can be found at `https://scheduleak.github.io/`.

### 3.8.1 Overriding Control Signals

**Attack Scenario and Objective:** A large number of real-time control systems encapsulate subsystems that control actuators. For instance, in modern automotive systems, the engine control unit (ECU) controls the valve in the electronic throttle body (ETB) to enable electronic throttle control (ETC). In most unmanned drones, the flight controller manages the rotary speed of the motors via the electronic speed controller (ESC). In these systems, the actuation signals such as PWM signals are periodically updated to guarantee a fast and consistent response for the control mission.

Let's consider an attacker who wants to be able to stealthily override the control in such systems – for the purpose of bad control by causing misbehavior or even taking over the control of the system for a short time span. To do so, the attacker gets into the system as a malicious task and tries to override the control signals. A brute force strategy of excessively

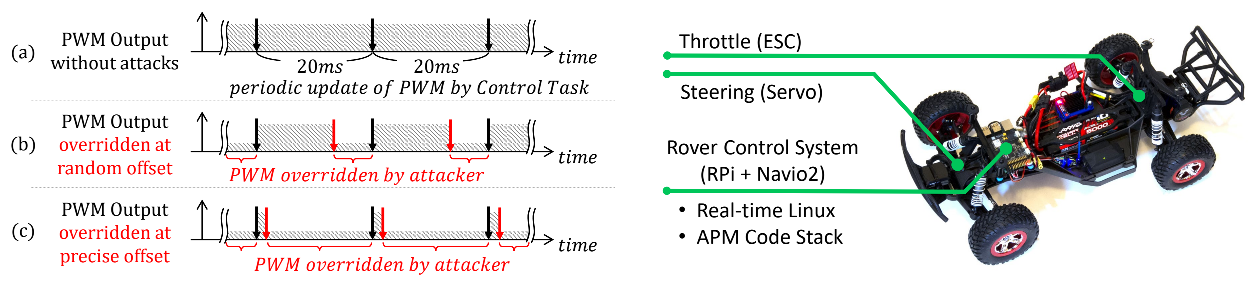

(a)
PWM Output without attacks
20ms
20ms
time
periodic update of PWM by Control Task
(b)
PWM Output overridden at random offset
time
PWM overridden by attacker
(c)
PWM Output overridden at precise offset
time
PWM overridden by attacker
Throttle (ESC)
Steering (Servo)
Rover Control System (RPi + Navio2)
• Real-time Linux
• APM Code Stack

Figure 3.23: An illustration of PWM channels on a rover system. (a) The PWM outputs are updated periodically by a $50Hz$ task. (b) A naive attack issuing the PWM updates at random instants may not be effective. (c) By carefully issuing the PWM updates right after the original updates, the PWM outputs can be overridden.

overriding the control signals will not work in this scenario because its high attack overhead can cause other real-time tasks to miss their deadlines and lead to a system crash. In this case, knowledge of exact timing when the control signals are updated and overriding them at the right instants allow the attacker to effectively take control with a low overhead.

**Implementation:** I implement this attack on a custom rover. Its control system is built with a Raspberry Pi 3 Model B board. A Navio2 module board that encapsulates various inertial sensors is attached to the Raspberry Pi board. The system runs Real-Time Linux (*i.e.,* Raspbian, kernel 4.9.45 with PREEMPT_RT patch) with Ardupilot [66] autopilot software suite (one of the most popular open-source code stack in the remote and autonomous control communities). It consists of a set of real-time and non-real-time tasks to perform control-related jobs such as refreshing GPS coordinates, decoding remote control commands, performing PID calculation and updating output signals. One of the tasks periodically updates the PWM values, with a period of $20ms$, for steering and throttle. The updates are sent over Serial Peripheral Interface (SPI) to the Navio2 module that outputs the PWM signals to a servo and a ESC. Figure 3.23(a) shows an illustration of the PWM output channels working under normal circumstances.

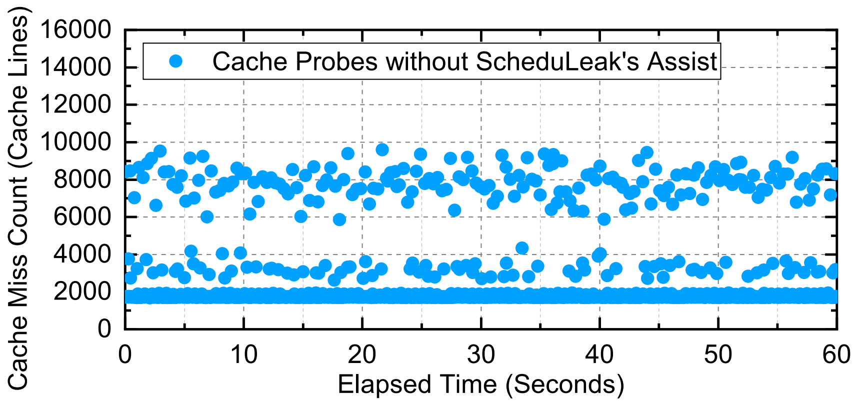
In this attack, I assume that the attacker has access to a low-priority, periodic task (as the observer task, $T_o = 50ms$) and a non-real-time Linux process (for launching the PWM overriding attack). The attacker's ultimate objective is to override the control signals updated by the victim task (*i.e.,* the $50Hz$ periodic task). In this implementation, the observer task uses a system call, `clock_gettime()`, to obtain clock counts (in nanoseconds) from `CLOCK_MONOTONIC`. Time measurement is further rounded up to microseconds when running the ScheduLeak algorithms since all task parameters are multiples of $1us$ in Ardupilot. Once the victim task's initial offset is determined, the attacker engages the non-real-time process to issue the PWM updates over the same interface that the victim task uses. Note that this is possible due to a lack of authentication between the Raspberry Pi board and the Navio2 module by design. This process keeps track of time by using `clock_gettime`() and issues two PWM updates (one for the steering and one for the throttle) whenever it determines that it has passed a victim task's arrival instant (*i.e.*, $t - \hat{\phi_v} \; mod \; T_v \geq 0$, where $t$ is the present time and $\hat{\phi_v}$ is the inferred victim task's initial offset). The process remains idle between two PWM updates to reduce the attack footprint.

**Attack Results:** Figures 3.23(b) and 3.23(c) show that the PWM output may be overridden using a different value to the PWM hardware. However, without exact schedule information, the attacker can only periodically send the updates with a randomly selected initial offset (Figure 3.23(b)). The random initial offset can be any point in the $20ms$ period. From the experiments, only the attack with an initial offset in the range between $\phi_v$ and $\phi_v + 8.3ms$ can produce an effective override of the steering and throttle controls. As a result, the attacker has a chance of 41.5% to select a valid initial offset and lead to an effective attack.

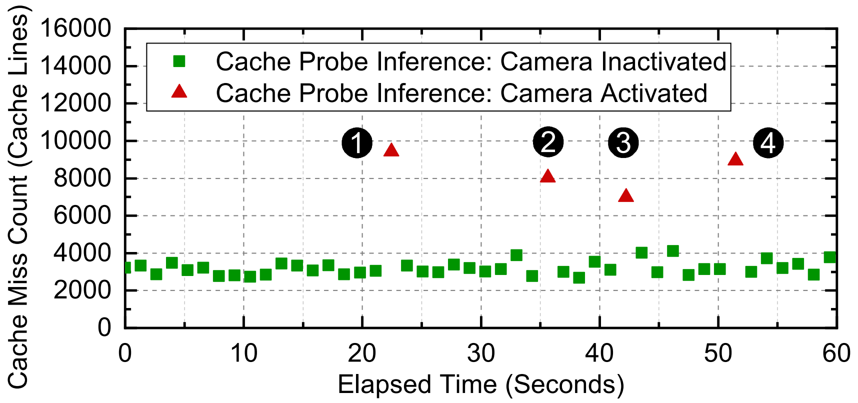
On the other hand, the attacker, after launching the ScheduLeak attack and knowing exactly when the victim task arrives, can carefully issue PWM update *right after the original update* to override the PWM output (Figure 3.23(c)). In this case, the attacker firstly runs the ScheduLeak algorithms in the observer task, yielding 0.9985 for the inference precision ratio (for inferring the victim task's initial offset) in a duration of 1 second. This allows the attacker to launch the PWM overriding attack in the non-real-time process with the precise inference of the victim task's initial offset. Note that an attacker's PWM update attempted at a victim task's arrival instant is executed after the victim task's job is finished (and hence after the original PWM update) since the non-real-time process has a priority lower than the victim task. Consequently, the attacker can take over control of the steering and throttle. By probing the PWM signals, I observe that the overridden PWM signals are active 85% of the time. As a result, I see that the rover no longer responds to the original control. Instead, the rover is driven by the attacker's commands. Since the attacker's task remains idle between two PWM updates, it takes up CPU utilization as small as 2.6%.

### 3.8.2 Inferring System Behaviors

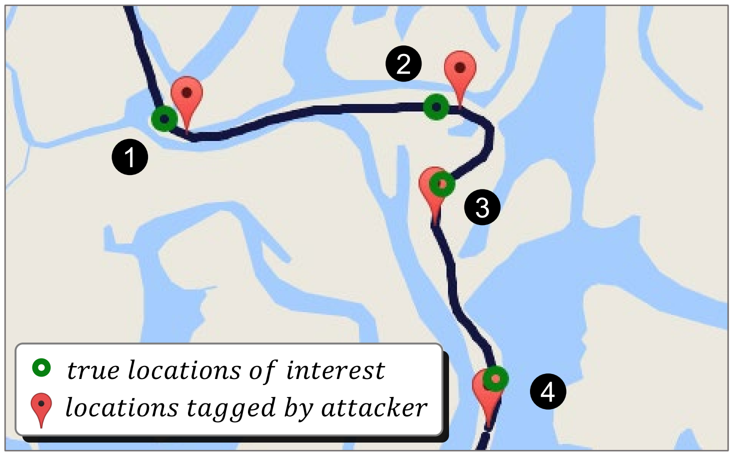
**Attack Scenario and Objective:** Let's consider a UAV system executing a surveillance mission. It captures high resolution images when flying over locations of high-interest. In this case, the attacker's goal is to extract the locations targeted by the UAV. The strategy



(a) Attack without ScheduLeak's assist.



(b) Attack with ScheduLeak's assist.



(c) HIL simulator recorded data.

Figure 3.24: Results of the cache-timing side-channel attacks in Section 3.8.2. (a) demonstrates that a random mechanism launching the attack at arbitrary instants will lead to many indistinguishable cache usage results. (b) shows a successful attack in which four camera activation events (numbered by 1 to 4) are identified from the cache probes using precise time information (inferred by ScheduLeak). (c) visualizes the UAV's trajectory (bold line), true locations-of-interest (green circles) and the attacker's inference (red pins) for the attack (b). The result shows that the attacker's inference matches the ground truth.

is to monitor when the surveillance camera on the UAV is switched to a execution mode in which high-resolution images are being processed. This can be done by exploiting a cache-timing side-channel attack to gauge the coarse-grained memory usage behavior of the task that handles the images. A high cache usage by this task would indicate that a high-resolution image is being processed; otherwise it would use less cache memory. However, a random sampling of the cache will result in noisy (and often useless) data since there exist other tasks in the system that also use the cache. In contrast, knowing when the task is scheduled to run allows the attacker to execute prime and probe attacks [67, 68] very close to the targeted task's execution.

**Implementation:** This attack is implemented in a hardware-in-the-loop (HIL) simulation with a Zedboard running FreeRTOS that simulates the control system on a UAV. The system consists of an image processing task (the victim task, $T_v = 33ms$) handling photos at a rate of $30Hz$ and four other tasks (unknown to the attacker) – all running in a periodic fashion. The victim task processes a large size of data when the UAV reaches a location of interest on a preloaded list. Other tasks consume differing amounts of memory. In this case, I assume that the attacker enters the system as the lowest-priority periodic task, $T_o = 40ms$. The attacker uses this task for both running the ScheduLeak algorithms and carrying out the cache-timing side-channel attack. The attacker's final goal is to observe the victim task's memory usage and learn the system behavior.

**Attack Results:** First, I consider an attacker who does not employ a ScheduLeak attack. The attacker launches the cache-timing side-channel attack during every period to try and estimate the cache usage of the victim. As shown in Figure 3.24(a), this produces many cache probes and it is hard to distinguish the cache usage of the victim task from other tasks. This results in an unsuccessful attack since no usage patterns from the victim task can be identified.

Next, let's consider the case in which the attacker leverages the ScheduLeak attack. In this case, the algorithms yield an inference precision ratio of 0.99 within a window of $3 \cdot LCM(T_o, T_v)$ (*i.e.,* 4 seconds). Then, the attacker is able to launch the cache-timing side-channel attack right before and after the victim is executed and skip those instants that are irrelevant. Figure 3.24(b) shows the result of the precise cache probe against the victim task. I see that the attack greatly reduces the noise caused by other tasks (96.9% of the cache probes are omitted) and is able to precisely identify the victim task's memory usage behavior. As a result, four camera activation instants can be identified from the spikes (red triangular points) shown in Figure 3.24(b). When coupled with the flight route information that the attacker obtains through other measures, it becomes possible to infer the locations of high-interest, as shown in Figure 3.24(c).

## 3.9 DISCUSSION

From the evaluation results, DyPS in EDF RTS performs well in inferring the victim task's phases and has similar performance characteristics to ScheduLeak in FP RTS under various task set conditions. As a result, given the same task set, running either FP scheduling or EDF scheduling does not seem to make a significant difference in terms of resisting the attacks. However, the DyPS algorithms do require an extra step to reconstruct the observer task's phase before proceeding to inferring the victim task's phase, which is the crucial point that distinguishes the scheduler side-channels in FP RTS and EDF RTS. As analyzed in Section 3.4.7, the observer task may suffer constant start time delays when there exist some harmonic tasks that have $\psi(\tau_o, \tau_i) = 1$ and $\phi_i$ aligned with $\phi_o$. Having such tasks may lead to an inaccurate $\widetilde{\phi_o}$ and further reduce the inference precision as evaluated in the experiments. This offers one potential effective measure for defending EDF RTS against the DyPS attack. Consequently, employing the EDF scheduling algorithm and adjusting the task parameters to satisfy the aforementioned conditions can be a simple yet cost effective defense. However, any change in the task parameters must fulfill both real-time requirements as well as the required performance. Thus, changing the task parameters may not always be applicable in RTS especially the legacy systems that are already deployed.

## 3.10 CONCLUSION

Successful security breaches in control systems (including cyber-physical systems) with real-time properties can have catastrophic effects. In many such systems, knowledge of the precise timing information of critical tasks could be beneficial to adversaries. The work presented in this chapter demonstrates how to capture this schedule timing information in a *stealthy* manner – *i.e.,* without being detected or causing any perturbations to the original system. This result answers the first key research question raised in Section 1.1 and validates my hypothesis with respect to the existence of the scheduler side-channels in preemptive RTS. Designers of such systems now need to be cognizant of such attack vectors and design the system to include countermeasures that can thwart potential intruders.

# CHAPTER 4: DIVERSIFICATION OF REAL-TIME SCHEDULES

In this chapter I intend to validate the hypothesis that it is possible to protect the RTS by *diversifying* the real-time schedule. By increasing randomness in the real-time schedule, its inferability is expected to reduce. Thus, even if an observer is able to record the exact schedule for a period of time, there is no guarantee that the next period will show the exact same order (and timing) of execution for the tasks.

## 4.1 INTRODUCTION

*Obfuscating the schedules*, *i.e.,* introducing randomness into the execution patterns of real-time tasks, could be one way to improve the security of RTS. This must be done in a careful manner, so as to not interfere with the timing guarantees that the system can provide, while still introducing diversity into the schedule. In this chapter I first introduce a schedule randomization protocol (Section 4.3.4) that I named REORDER (REal-time ObfuscateR for Dynamic SchedulER). I achieve this by using *bounded priority inversions* at run-time (see Section 4.3.3 for more details). REORDER obfuscates the earliest deadline first (EDF) scheduling policy; EDF is a dynamic task scheduler that can, theoretically, utilize a CPU to its fullest. It is widely supported by many real-world RTS and operating systems, *e.g.,* Erika Enterprise [69], RTEMS [70], *etc.* and even Linux [65]. Existing work on protecting real-time schedulers [49, 51] is *(a)* focused on static scheduling algorithms and *(b)* inadequate for measuring the effects of obfuscation. Obfuscating the schedules for dynamic priority algorithms such as EDF, to achieve a high level of randomization, is a much harder proposition than that for static algorithms. One important problem is how to bound the time allocated for allowing priority inversions since the job deadlines dynamically change as the execution proceeds[1]. REORDER guarantees that if a given real-time system was schedulable (*i.e.,* meets all of its timing and deadline constraints) by the vanilla EDF scheduler, then the *obfuscated schedule will also meet the same guarantees.*

While the goal of the protocol is to increase as much randomness as possible in the schedule, it is restricted when the task set utilization is high due to the tight real-time constraints, making it less effective against scheduler side-channel attacks. Additionally, it does not offer any security guarantee with respect to the degree of protection. As a result, a system designer employing TaskShuffler [49] or REORDER may not get the desired protection against the scheduler side-channels.

[1]In static algorithms, these bounds can easily be computed offline and stored in lookup tables.

To address the problem of the scheduler side-channels, I propose the notion of "*schedule indistinguishability*" that captures the difficulty of identifying a task/job from another in the schedule. I then propose an $\epsilon$-Scheduler that implements the schedule indistinguishability. At a high level, my goal is to abate the determinism and periodicity by adding sufficiently large noise to the task schedules, albeit with strong analytical guarantees to achieve a desired schedule indistinguishability. This is strategy is similar to differential privacy in the context of databases where Laplace distribution-based noise is added to statistical query outputs. In $\epsilon$-Schedulers, noise derived from bounded Laplace distribution is added to the task's inter-arrival times to obscure the periodicity embedded in the task schedule. It creates diversified schedules that protect the tasks from the scheduler side-channels. Such a scheduler is particularly suitable for the applications and systems that can tolerate deadline misses and varied execution frequencies. As a demonstration for the type of systems to which $\epsilon$-Scheduler is applicable, I conduct experiments on a real platform running a real application, presented in Section 4.7.6.

## 4.2 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 $\{1, 2, ..., n\}$. I denote the Laplace distribution with location $\mu$ and scale $b$ and Lap($b$) by Lap($\mu, b$) and I write Lap($b$) when $\mu = 0$. For a random variable $x$, taking values from a Laplace distribution is denoted by $x \sim$ Lap($\cdot$). For convenience, I sometimes abuse notation and denote a random variable $x \sim$ Lap($\cdot$) simply by Lap($\cdot$).

I consider a discrete time model [62]. In this context, I mainly focus on the issue that is concerned with the timing in a single RTS. I 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. I 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 $[a, b - 1]$.

## 4.3 THE REORDER SCHEDULER

### 4.3.1 System and Adversary Model

Let us consider the problem of scheduling a set of $n$ periodic tasks $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$ on a single processor, using the EDF scheduling policy. For simplicity of notation, I use the same symbol $\tau_i$ to denote a task's jobs and use the term *task* and *job* interchangeably. I also denote $d_i$ as the absolute deadline of $\tau_i$ (*i.e.,* deadline of any given job of $\tau_i$). I assume cache related preemption delay is negligible compared to WCET of the tasks. I do not consider any precedence or synchronization constraints among tasks and $C_i, T_i, D_i \in \mathbb{N}^+$,. I further assume that the tasks have constrained-deadlines, *i.e.,* $D_i \leq T_i$ and the task set is schedulable by the EDF scheduling policy, that is the worst-case response time (WCRT) of each task is less than its deadline – since REORDER will be trivially ineffective for an unschedulable task set.

Under the periodic task model, the schedule produced by any preemptive scheduling policy, for a periodic task set, is cyclic *i.e.,* the system will repeat the task arrival pattern after an interval that coincides with the task set's hyper-period , denoted by $L$.

I assume that the attackers have access to the timing parameters of the task sets and also have knowledge of which scheduling policy is being used. The adversary's objective is to get detailed information about the execution patterns of the real-time tasks and cause greater damage, to the system by exploiting the precise schedule information.

The attacker may exploit some side-channels (*e.g.,* power consumption, schedule preemptions, electromagnetic (EM) emanations and temperature) to observe and reconstruct the system schedule. A smart attacker possessing sufficient system information can carry out more advanced attacks under the right conditions, to move the system to an unsafe state. For example, in the now famous Stuxnet attack [1, 61], the malware was remnant in the system for *months* to collect sensitive information before the main attack. It is possible for a denial-of-service attack to target only a specific service handled by a critical task when the precise schedule information is obtainable.

A side-channel attack [68, 71] is also another typical class of attacks that can benefit from such schedule reconstruction attacks. For example, one of the case studies presented in Section 3.8 shows that the precise schedule information can be exploited to assist in determining the *prime* and *probe* [67, 68] instants in a cache side-channel attack to increase the chance of success [20].

I further assume that the scheduler is not compromised and the attacker does not have access to the scheduler. Without this assumption, the attacker can undermine the scheduler or directly obtain the schedule information. The objective of this work, then, is to reduce the inferability of the schedule for real-time tasksets (and also reduce possibility of other attacks that depend of predictable schedules) while meeting real-time guarantees. The randomness introduced to the schedule increases variations in the system and hence makes attacks that rely on the determinism of the real-time schedule, harder.

### 4.3.2 Overview of REORDER

The focus of the design of REORDER is such that, even if an observer is able to capture the exact schedule for a period of time (for instance, for a few hyper-periods), REORDER will schedule tasks in a way that succeeding execution traces will show different orders (and timing) of execution for the tasks. The main idea is that at each scheduling point, the scheduler *pick a random task from the ready queue* and schedule it for execution. However such random selection may lead to priority inversions [72] and any arbitrary selection may result in *missed deadlines* – hence, putting at risk the safety of the system. REORDER solves this problem by allowing *bounded priority inversions*. It restricts how the schedule may use priority inversions without violating real-time constraints (*e.g.,* deadline) of the tasks. To ensure this, REORDER calculates an "acceptable" priority inversion budget. If the budget is exhausted during execution, then I stop allowing lower priority tasks to execute ahead of the higher priority task that has the empty budget.

### 4.3.3 Randomization with Priority Inversion

A key step that is necessary for randomization is to calculate the maximum amount of time that lower priority jobs, $lp(\tau_i)$, can execute before $\tau_i$. This is much harder in EDF compared to the fixed-priority system (that prior work, TaskShuffler [49], was focused on) due to the dynamic nature of EDF (*i.e.,* the task priority varies at run-time). Therefore I define the *worst-case inversion budget* (WCIB) $V_i$ that represents the maximum amount of time for which a job of some task $\tau_i$ with relative deadline $d_i$ may be blocked by a job of some task $\tau_j \in \Gamma, j \neq i$ with $d_j > d_i$ (and hence lower relative priority than $\tau_i$). In the following I illustrate how I calculate WCIB for each task by utilizing the response time analysis [73, 74] for EDF.

**Bounding Priority Inversions.** The WCRT of $\tau_i$ is the maximum time between the arrival of a job of $\tau_i$ and its completion. Our idea of bounding priority inversions is to calculate the slack times for each task (*e.g.,* difference between deadline and response time) and allow low priority tasks to execute up to that amount of time. I therefore define the WCIB of $\tau_i$ as follows:

$$V_i = D_i - R_i. \tag{4.1}$$

where $R_i$ represents an upper bound of WCRT. *The $V_i$ represents the maximum amount of time for which all lower priority jobs $lp(\tau_i)$ (e.g., $d_j > d_i$) are allowed to execute while an instance of $\tau_i$ is still unfinished without missing its deadline, even in the worst-case* *scenario.* The REORDER protocol guarantees that the real-time constraints are satisfied by bounding priority inversions using $V_i, \forall \tau_i \in \Gamma$. Note that WCIB can be negative for some $\tau_i$ – although non-positive WCIB does not attribute that the taskset is unschedulable. At each scheduling point $t$, our idea is to execute some low priority job $\tau_j$ with $V_j > 0$ up to $\min(\widehat{C}_j^t, V_j)$ additional time-units before it leaves the processor for highest priority job where $\widehat{C}_j^t$ represents the remaining execution time of $\tau_j$ at $t$.

I enforce the WCIB at run-time by maintaining a per-job counter, *remaining inversion budget* (RIB) $v_i$, $0 \leq v_i \leq V_i$. RIB is initialized to $V_i$ upon each activation of the jobs of $\tau_i$ and decremented for each time unit when $\tau_i$ is blocked by any lower priority job. When $v_i$ reaches zero no job with absolute deadline greater than $d_i$ is allowed to run until $\tau_i$ completes. Note that not all the jobs of $\tau_i$ may need $C_i$ time unit for computation (recall that $C_i$ is the worst-case bound of the execution time). If some low-priority job $\tau_j$ (*e.g.,* $d_j > d_i$) that blocks $\tau_i$ finishes earlier than $C_j$, the RIB $v_i$ will not be decreased accordingly.

For a given non-negative WCIB, jobs of $\tau_i$ can be delayed for up to $V_i$ by priority inversions. The WCRT of $\tau_i$ is bounded by $R_i + V_i = R_i + D_i - R_i = D_i$. Hence, $\tau_i$ is schedulable with the REORDER protocol and I can assert the following:

**Proposition 4.1.** If $\Gamma$ is schedulable under EDF, WCIB is non-negative for some $\tau_i$ and low priority jobs of $\tau_i$ do not delay $\tau_i$ more than $V_i$ then REORDER will not violate the real-time constraints of $\tau_i$.

**Selection of Candidate Jobs for Randomization.** As mentioned earlier, when the run-time counter RIB (*i.e.,* $v_i$) reaches zero, no jobs with deadline greater than $d_i$ can run while $\tau_i$ has an outstanding job. However, lower priority jobs could cause $\tau_i$ to miss its deadline by inducing the worst-case interference from the higher priority jobs, *i.e.,* $\forall d_j < d_i$, due to the chain reaction. Therefore, to preserve the schedulability of such jobs I must prevent it from experiencing such additional delays. I achieve this by the following "*randomization priority inversion policy*" (RPIP):

*If RIB $v_i < 0$ for some $\tau_i \in \Gamma$, no job $\tau_j$ with $d_j > d_i$ is allowed to run while any of high priority job $\tau_k$ with $d_k < d_i$ has an unfinished job.*

In order to enforce RPIP at run-time, at each scheduling decision point, I now define the variable minimum inversion deadline $m_i^t$ for jobs of $\tau_i$ as follows:

$$m_i^t = \min\{d_j | \tau_j \in \mathcal{R}_{\mathcal{Q}}^t,\ d_j \geq d_i \wedge v_j \leq 0\}. \tag{4.2}$$

where $\mathcal{R}_{\mathcal{Q}}^t$ is the ready queue at scheduling point $t$. When there is no such task as $\tau_j$, $m_i^t$ is set to an arbitrarily large (*e.g.,* infinite) deadline. The variable $m_i^t$ allows us to determine which jobs to *exclude* from priority inversions. That is, no job that has a higher deadline than $m_i^t$ can be scheduled as long as $\tau_i$ has an unfinished job. Otherwise, the job with relative deadline $m_i^t$ (not the job $\tau_i$) could miss its deadline.

**Example 4.1.** The taskset $\Gamma_{ex1} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ contains the following parameters:

Table 4.1: An RTS task set of 4 periodic tasks.

| | $C_i$ | $T_i = D_i$ | $V_i$ |
|---|---|---|---|
| $\tau_1$ | 4 | 10 | 1 |
| $\tau_2$ | 1 | 20 | $-2$ |
| $\tau_3$ | 1 | 5 | $-2$ |
| $\tau_4$ | 2 | 12 | $-1$ |

At $t = 0$, $d_1 = 10$, $d_2 = 20$, $d_3 = 5$, $d_4 = 12$. For notational convenience, let us denote $m_i^0$ as $m_i$. Hence $m_1 = 12$, $m_2 = \infty$, $m_3 = 12$ and $m_4 = 20$. Therefore at $t = 0$ the job $\tau_2$ and $\tau_4$ are not allowed to participate in priority inversion (since $d_2, d_4 > m_i, i \in \{1, 3\}$ and $\tau_1$, $\tau_3$ have not completed.

It can be shown that at any scheduling point $t$ I can enforce RPIP by only examining the inversion deadline of highest priority (*e.g.,* shortest deadline) job, $m_{HP}^t$ [49]. Hence, at each scheduling decision, REORDER excludes all ready jobs from the selection that have higher deadline than $m_{HP}^t$.

### 4.3.4 The Randomization Protocol

The REORDER protocol selects a new job using the following sequence of steps at every scheduling decision point.

1. *Candidate Selection:* At each scheduling point $t$, the REORDER protocol searches for possible candidate jobs (that can be used for priority inversion) in the ready queue. Let us denote $\mathcal{R}_Q^t$ as the set of ready jobs, $\tau_{HP} \in \mathcal{R}_Q^t$ is the highest priority (*i.e.,* shortest deadline) job in the ready queue and $\mathcal{C}_{\mathcal{L}}^t$ represents the set of candidate jobs at some scheduling point $t$.

    - I first check the RIB of the highest priority job $\tau_{HP} \in \mathcal{R}_Q^t$. If the RIB is zero, then $\tau_{HP}$ is added to the candidate and REORDER moves to Step 2 since priority inversion is not possible due to its inversion budget being non-positive.

- When RIB is non-negative (*i.e.,* $v_{HP} > 0$), I iterate through the ready queue and add the job $\tau_i \in \mathcal{R}_{\mathcal{Q}}^t$ to the candidate list $\mathcal{C}_{\mathcal{L}}^t$ if its deadline is less than or equal to $m_{HP}^t$ (*i.e.,* the minimum inversion deadline of the highest-priority job at scheduling point $t$).

2. *Randomizing the Schedule:* This step selects a random job from the ready queue for execution. The selected job will run until the next scheduling decision point $t'$. I randomly pick a job $\tau_R$ from $\mathcal{C}_{\mathcal{L}}^t$ and set the next scheduling decision point as follows:

   - If $\tau_R$ is the highest priority job in the ready queue, the next decision point $t'$ will be either when the job finishes or a new job of another task arrives.
   - Otherwise, the next decision[2] will be made at when $\tau_R$ completes or the inversion budget expires, that is,

$$t' = t + \min(\overbrace{\widehat{C}_R^t}^{\text{remaining execution time of } \tau_R}, \underbrace{\widehat{v}}_{\text{maximum remaining priority inversion budget}}) \tag{4.3}$$

   unless a new job arrives before time $t'$ where $\widehat{v} = \min(v_j | \tau_j \in \mathcal{R}_{\mathcal{Q}}^t \wedge d_j < d_R)$ and $\widehat{C}_R^t$ represents the remaining execution time of $\tau_R$. Note that the variable $\widehat{v}$ is always positive since every job with a higher priority than the selected job has some remaining inversion budget. Otherwise, $\tau_R$ would not have been added to the candidate list in Step 1.

Algorithm 4.1 formally presents the proposed schedule randomization protocol. This event-driven algorithm executes at the scheduler-level and takes the task set (with idle time) $\Gamma' = \Gamma \cup \{\tau_I\}$ as an input. At each scheduling decision point $t$, a ready job is (randomly) selected for scheduling and the next scheduling decision point $t'$ is determined.

In Lines 1-7, the algorithm first selects the set of candidate jobs $\mathcal{C}_{\mathcal{L}}^t$ using the procedure described in Section 4.3.4 (see Step 1). If the highest priority job $\mathcal{R}_{\mathcal{Q}}^{HP}$ has negative inversion budget (*e.g.,* $v_{HP} \leq 0$), it will be scheduled for execution (Line 10). Otherwise it schedules a random job from the candidate list (Line 14). If the selected job is the highest priority job, the next scheduling point $t'$ is set when the job completes or a new job of another task arrives (Line 14 and 19). If the selected job is not the highest priority one, the algorithm selects $t'$ when the current inversion budget expires, unless the job completes or a new job arrives before $t'$ (Line 20).

[2]Section 4.3.5 presents another approach to trigger scheduling decisions.

**Algorithm 4.1:** Schedule Randomization Protocol

```
Input: Augmented task set $\Gamma' = \Gamma \cup \{\tau_I\}$ and current scheduling point $t$
Output: The randomized schedule $S_t$ and the next scheduling point $t'$
1  $\mathcal{R}_{\mathcal{Q}}^t :=$ set of ready jobs
2  Add the highest priority job to the candidate list, i.e., $\mathcal{C}_{\mathcal{L}}^t := \{\mathcal{R}_{\mathcal{Q}}^{HP}\}$
3  /* Search candidate jobs if the highest priority job has non-zero inversion budget */
4  if $v_{HP} > 0$ then
5      foreach $\tau_j \in \mathcal{R}_{\mathcal{Q}}^t$ do
6          if $d_j \leq m_{HP}^t$ then
7              $\mathcal{R}_{\mathcal{Q}}^t := \mathcal{R}_{\mathcal{Q}}^t \cup \{\tau_j\}$ /* add $\tau_j$ to candidate list */
8  if $\mathcal{C}_{\mathcal{L}}^t = \{\mathcal{R}_{\mathcal{Q}}^{HP}\}$ then
9      /* schedule the highest priority (shortest deadline) job */
10     $S_t := \mathcal{R}_{\mathcal{Q}}^{HP}$
11     Set next scheduling point $t' :=$ when new job arrives or current job completes
12 else
13     /* randomly select a job $\tau_R$ from $\mathcal{C}_{\mathcal{L}}^t$ */
14     $S_t := \tau_R$
15     if $\tau_R = \mathcal{R}_{\mathcal{Q}}^{HP}$ then
16         Set next scheduling point $t' :=$ when new job arrives or current job completes
17     else
18         /* set the next random scheduling point $t'$ as a function of current job
             completion or budget expiration time (unless a new job arrives before $t'$) */
19         $\Delta t := \text{rand}(1, \min(\widehat{C}_R^t, \widehat{v}))$
20         Set next scheduling point $t' := t + \Delta t$
21 /* return the scheduled job and the next scheduling point */
22 return $(S_t, t')$
```

The algorithm iterates over the jobs in the current ready queue $\mathcal{R}_{\mathcal{Q}}^t$ once and makes a single draw from the candidate list $\mathcal{C}_{\mathcal{L}}^t \subseteq \mathcal{R}_{\mathcal{Q}}^t$. Assuming a single draw from a uniform distribution (Line 14 and 19) takes no more than $O(|\mathcal{R}_{\mathcal{Q}}^t|)$, the complexity of each instance of the algorithm is $O(|\mathcal{R}_{\mathcal{Q}}^t|)$ .

### 4.3.5 Randomization Modes

**Unused Time Reclamation.** As mentioned earlier, not all the jobs of a task may require worst-case unit of time for its computation. I propose to reclaim this unused time (*e.g.,* difference between WCET and actual execution time) to increase the inversion budget for lower priority jobs. In the case that the (randomly) selected job finishes earlier (*i.e.,* the actual execution time is smaller than its WCET), the unused time that is reserved for this job can be transferred to its lower priority jobs (*i.e.,* those ready jobs that have higher deadlines at the moment) as extra inversion budget. Therefore, when enabling this feature, the RIBs of the lower priority jobs are updated (at the scheduling point $t'$ when the selected job $\tau_R$ finishes its execution) as follows: $v_j = v_j + \delta_R^{t'},\ \tau_j \in \mathcal{R}_Q^{t'} \wedge d_j > d_R$ where $\delta_R^{t'}$ represents the unused time over WCET and $\mathcal{R}_Q^{t'}$ is the ready queue at time $t'$. Note that the real-time constraints (*i.e.,* deadlines) are respected since Eq. 4.1 for every ready job at time $t'$ still holds (*i.e.,* $V_j + \delta_R^{t'} = D_j - (R_j - \delta_R^{t'})$) with the unused time transferring.

When there are no tasks in the ready queue (*e.g.,* during slack time), the processor is *idle*, *e.g.,* nothing is executing in the system. Although REORDER brings variations between the hyperperiods when compared to the vanilla EDF, randomizing only real-time tasks results in the schedule being somewhat predictable since the idle times (*i.e.,* slack) appear in nearly same slots. I address this problem by *scrambling the idle times* along with the real-time tasks in the next section.

**Idle Time Scheduling.** One of the limitations of randomizing only the tasks is that the task execution is squeezed between the idle time slots and the latter remain predictable. The work-conserving nature of EDF causes separations between task executions and idle times. Hence some tasks appear at similar places over multiple hyperperiods. One way to address this problem and improve schedule randomness is to *idle the processor*, intentionally, at random times [49]. I achieve this by considering idle times as instances of an additional task, referred to as the *idle task*, $\tau_I$. Then, the randomization protocol can be applied over the *augmented taskset* $\Gamma' = \Gamma \cup \{\tau_I\}$.

It can be noted that $\tau_I$ has infinite period, deadline and execution time, and hence always executes with the *lowest* priority. Hence $\tau_I$ can force all other tasks $\tau_i \in \Gamma$ to maximally consume their inversion budgets. During randomization the idle task will convert a work-conserving schedule to a non-work-conserving one, but it will not cause any starvation for other tasks. This is because Step 2 of the REORDER protocol (see Section 4.3.4) selects candidate tasks in a way that real-time constraints for *all* tasks in the system will always be respected. Randomizing the idle task effectively makes tasks appear across wider ranges and thus reduces predictability. As a result, the schedule can be less susceptible to attacks that depend on the predictability of RTS.

**Fine-Grained Switching.** In TaskShuffler [49] I proposed to decrease the inferability of the fixed-priority scheduler by randomly *yielding* a job, early, during execution. As a result the schedule will be fragmented at different time-points and thus will bring more variations across execution windows. The proposed REORDER protocol can also be modified to incorporate such a feature. Recall that the scheduling decisions in my scheme are made either when: *(i)* a new job arrives, *(ii)* a job completes, or *(iii)* the inversion budget expires (refer to Step 2 in Section 4.3.4). Therefore I can achieve fine-grained switching by modifying the next scheduling decision point $t'$ in Eq. (4.3) as follows: $t' = t + \mathrm{rand}(1, \min(\widehat{C}_R^t, \widehat{v}))$ where the function $\mathrm{rand}(a, b)$ outputs a random number between $[a, b]$.

## 4.4 THE $\epsilon$-SCHEDULER

### 4.4.1 System and Adversary Model

In this work, I consider a single processor, preemptive real-time system in which deadline misses are tolerable [75, 76]. The system contains a task set consisting of $N$ real-time tasks $\Gamma = \{\tau_i \mid i \in [N]\}$, schedulable by a dynamic-priority scheduler (*e.g.,* an Earliest Deadline First, EDF, scheduler [9]). I assume the real-time tasks are independent (*i.e.,* no dependencies between tasks). A real-time task can be a periodic task (that has a fixed period) or a flexible task (that has flexible period choices within a predefined range)[3] [77]. I model a real-time task $\tau_i$ 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 $\eta_i$ is a *task inter-arrival time function* to be defined below. It can be easily seen that a periodic task is a flexible task with the choice of periods limited to a constant value. That is, $\mathcal{T}_i = \{T_{i,1}\}$ when $\tau_i$ is a periodic task and I sometimes use $T_i$ to denote such a fixed period for a task for simplicity. A task's execution instance is aborted upon missing its current deadline and it shall not impact the release of the task's next execution instance. Under this assumption, I further assume that applications running on the real-time tasks should be able to work with, and tolerate, the dynamic changes in the task's run-time frequencies.

To easier formulate the problem and better profile the task's periodicity, let's assume the system behaves deterministically and thus a task's execution behavior *w.r.t.* its inter-arrival time sequence is modeled by a *task inter-arrival time function* (each task has a dedicated function.)

**Definition 4.1.** (Task Inter-Arrival Time Function.) For a task $\tau_i$, its task inter-arrival time function is defined as

$$\eta_i : \mathbb{N} \to \mathcal{T}_i \tag{4.4}$$

[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, I intentionally exclude these types of tasks in the task model to be focused on the periodic components.

and $\eta_i(j)$ gives the task's inter-arrival time at $j^{th}$ instance. The resulting inter-arrival time is a value 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 constant 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 scheduling 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 a task $\tau_i$ arrives as the $j^{th}$ instance, the scheduler obtains its currently desired period from $\eta_i(j)$ and configures the current absolute deadline as well as the next consecutive arrival time accordingly.

I'm mainly concerned about the scheduler side-channels that are exposed by the periodic nature of RTS. I assume that an adversary observes the system schedule via some existing side-channels such as power consumption traces [15], schedule preemptions [20, 44], electromagnetic (EM) emanations [16] and temperature [17]. I further assume that the scheduler is not compromised and 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.

Some existing attacks have demonstrated that the periodicity can be exploited to learn a targeted task's execution state which can be used to launch further attacks causing greater damages to the system with a high precision [20, 48]. These types of attacks rely on the fact that periodicity exists in the real-time tasks being targeted. In this work, I aim to eliminate scheduler side-channels by obscuring the task periodicity in the schedule. To this end, my goal in this work can be seen as to achieve *schedule indistinguishability* in the system and depending on the attacker's intent it can be further defined and categorized into two *(i) job-level indistinguishability* and *(ii) task-level indistinguishability*:

- **Job-level indistinguishability.** The *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 earlier, 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 [77]. Leaking which period the control task is running at 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 period.

- **Task-level indistinguishability.** On the other hand, the *task-level indistinguishability* refers to the difficulty of distinguishing a task from another in a schedule. In an

RTS in which all tasks are strictly periodic, it is generally not hard to distinguish and identify each individual task's period from a schedule (see Section 4.7.1 for an exemplified analysis). As a result, tasks are at risk leaking critical information. For instance, in the ScheduLeak attack [20], the adversary exploits the periodicity to extract the execution behavior of a target real-time task from an observed schedule. Achieving a task-level indistinguishability weakens the adversary's ability to learn information about a specific task from the schedule.

It's intuitive to see that the job-level indistinguishability is a necessary condition for the task-level indistinguishability. That is, if the task-level indistinguishability can be achieved, the job-level indistinguishability for each task 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 for each task we can tweak. In such a case, the task-level indistinguishability may be infeasible to achieve. In this work, I propose an extended task model and a real-time scheduler with an inter-arrival time randomized mechanism to achieve the job-level indistinguishability and, when feasible, the task-level indistinguishability.

### 4.4.2 Randomizing Inter-Arrival Times

Let's consider a task $\tau_i$ and its inter-arrival time function $\eta_i$. The function produces a constant inter-arrival time (*i.e.,* period) if the task is strictly periodic for a given duration. Such consistent inter-arrival times constitute periodicity within the given duration. To break its periodicity, I intend to randomize each inter-arrival time. To this end, I propose a *inter-arrival time randomized mechanism*, denoted by $\mathcal{R}(\cdot)$, that is placed in the scheduler to generate a randomized version of the inter-arrival time obtained from $\eta_i$ by adding random noise. The inter-arrival time randomized mechanism is defined as

$$\mathcal{R}(\tau_i, j) = \lfloor \overbrace{\eta_i(j)}^{\text{the } j^{th} \text{ inter-arrival time of the task } \tau_i} + \underbrace{Y}_{\text{random noise drawn from some distribution centered at } 0} \rceil \tag{4.5}$$

where $\tau_i \in \Gamma, j \in \mathbb{N}$ represent that the $j^{th}$ inter-arrival time of the task $\tau_i$ is being generated and $Y$ 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

$\epsilon$-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 the task's periodicity. Yet, without specifying a noise distribution, it may be insufficient to obscure the task's periodicity, for example, when the noise's variance is insignificant. In advance of examining the noise addition mechanism, I first formally define the indistinguishability in our context.

### 4.4.3 Inter-Arrival Time Indistinguishability

As introduced in Section 4.4.1, I'm concerned with the job/task-level indistinguishabilities. To provide guarantees for such indistinguishabilities with the defined randomized mechanism, the inter-arrival time indistinguishability that's similar to the notion of differential privacy [23, 24] is used.

**Definition 4.2.** ($\epsilon$-Indistinguishability Inter-Arrival Time Randomized Mechanism.) An inter-arrival time randomized mechanism $\mathcal{R}(\cdot)$ is $\epsilon$-indistinguishable if

$$\Pr[\overbrace{\mathcal{R}(\tau, j)}^{\text{any randomized inter-arrival time of any given task } \tau} \in \mathcal{S}] \leq e^{\epsilon}\Pr[\underbrace{\mathcal{R}(\tau', j')}_{\text{any randomized inter-arrival time of any given task } \tau'} \in \mathcal{S}] \tag{4.6}$$

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 the inter-arrival time indistinguishability for a single job instance if Equation 4.6 is satisfied.

Note that Definition 4.2 is general enough to consider both the job-level and task-level indistinguishabilities. When $\tau \neq \tau'$, the task-level indistinguishability is implied; when$\tau = \tau'$, the job-level indistinguishability is implied. It is worth noting that we can maintain an independent $\epsilon_i$ value for each task $\tau_i$ and each of them achieves their own $\epsilon_i$-indistinguishability. The indistinguishability for the whole task set is determined by the worst of the $\epsilon_i$ [78] (which corresponds to the task-level indistinguishability).

### 4.4.4 Inter-Arrival Time Sensitivity and Laplace Noise

To determine the degree of noise to be added to make two inter-arrival times indistinguishable, I define a *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).

**Definition 4.3.** (Inter-Arrival Time Sensitivity.) The inter-arrival time sensitivity reflects the sensitivity of the function $\eta_\tau(\cdot)$. The sensitivity is defined depending on the desired indistinguishability goal:
*(i)* Job-level indistinguishability: the inter-arrival time sensitivity for the job-level indistinguishability, denoted by $\Delta\eta_\tau$ for a given task $\tau$, is defined as

$$\Delta\eta_\tau =: \max_{\substack{j,j' \in \mathbb{N} \\ j \neq j'}} |\, \overset{\text{distance between any two inter-arrival times of the task } \tau}{\boxed{\eta_\tau(j) - \eta_\tau(j')}} \,| \tag{4.7}$$

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

$$\Delta\eta_\Gamma =: \max_{\substack{\tau,\tau' \in \Gamma \\ j,j' \in \mathbb{N}}} |\, \overset{\text{distance between any two inter-arrival times of any two tasks in the task set } \Gamma}{\boxed{\eta_\tau(j) - \eta_{\tau'}(j')}} \,| \tag{4.8}$$

that is task-set-dependent.

For simplicity, I use $\Delta\eta$ to represent either of the sensitivities when the context is clear. Then, the use of the Laplace distribution $\mathrm{Lap}(\eta_\tau, \frac{\Delta\eta}{\epsilon})$ for generating the randomized inter-arrival times preserves $\epsilon$-indistinguishability defined in Definition 4.2 for a single job instance. This property can be easily proved by expanding Equation 4.6 with the probability density function of the $\mathrm{Lap}(\eta_\tau, \frac{\Delta\eta}{\epsilon})$ distribution [24, 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 $\Delta\eta = \Delta\eta_\Gamma$.

### 4.4.5 $\epsilon$-Indistinguishability in J Instances

The randomized mechanism $\mathcal{R}(\cdot)$ with Laplace noise $\mathrm{Lap}(\frac{\Delta\eta}{\epsilon})$ offers $\epsilon$-indistinguishability for a single instance. However, an attacker typically observes a longer schedule. Therefore, I'm more interested in the conditions for achieving $\epsilon$-indistinguishability for a certain duration.

As a noise draw occurs in every job instance, based on the theorem of Sequential Composition [78, 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 same level of indistinguishability for a certain duration. To this end, I measure the duration in the number of job instances (which corresponds to the number of noise draws for the corresponding inter-arrival times). Then, I use the following theorem to determine the scale of the noise for preserving $\epsilon$-differential privacy up to $J$ job instances.

**Theorem 4.1.** The Laplace randomized mechanism $\mathcal{R}(\cdot)$ with the scale $\frac{J\Delta\eta}{\epsilon}$ is $\epsilon$-indistinguishable up to $J$ job instances.

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

$$\Pr[\mathcal{R}^J(\tau, j) \in \mathcal{W}] \leq e^{\epsilon}\Pr[\mathcal{R}^J(\tau', j') \in \mathcal{W}] \tag{4.9}$$

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

$$\Pr[\mathcal{R}^J(\tau, j) = w] = \prod_{k\in[J]} \Pr[\mathcal{R}(\tau, j + k - 1) = \omega_k] \tag{4.10}$$

where $\omega_k$ is calculated by $\eta_\tau(\cdot) + \text{Lap}(b)$ in which $b$ is the Laplace distribution parameter. Expanding with the probability density function given in Equation 2.1, the right term in the above equation can be rewritten as

$$\prod_{k\in[J]} \frac{1}{2b}\exp(-\frac{|\omega_k - \eta_\tau(j + k - 1)|}{b}) \tag{4.11}$$

Then

$$\frac{\Pr[\mathcal{R}^J(\tau, j) = w]}{\Pr[\mathcal{R}^J(\tau', j') = w]} = \prod_{k\in[J]} \frac{\frac{1}{2b}\exp(-\frac{|\omega_k - \eta_\tau(j+k-1)|}{b})}{\frac{1}{2b}\exp(-\frac{|\omega_k - \eta_{\tau'}(j'+k-1)|}{b})} \tag{4.12}$$

$$= \prod_{k\in[J]} \exp(\frac{|\eta_\tau(j + k - 1) - \eta_\tau(j' + k - 1)|}{b}) \tag{4.13}$$

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 4.3). The above becomes

$$\prod_{k\in[J]} \exp(\frac{\Delta\eta}{b}) = \exp(\sum_{k\in[J]} \frac{\Delta\eta}{b}) \tag{4.14}$$

$$= \exp(\frac{J\Delta\eta}{b}) \tag{4.15}$$

Using Equation 4.9, we can derive $b$ from

$$\exp(\frac{J\Delta\eta}{b}) \leq \exp(\epsilon) \Rightarrow b \geq \frac{J\Delta\eta}{\epsilon} \tag{4.16}$$

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

The assignment of $J$ for a given task set is discussed in Section 4.4.9.

### 4.4.6 Bounded Laplace Randomized Mechanism

While the introduced Laplace randomized mechanism offers $\epsilon$-indistinguishability, the unbounded output domain for the randomized inter-arrival times makes it infeasible to adopt in real systems. To address this problem, I introduce the bounded Laplace randomized mechanism. That is, 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 [79]. Truncation is to project values outside the domain to the closest value within the domain. Bounding, used in this work, 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 randomized mechanism to return randomized inter-arrival times within a range that's feasible for the given underlying scheduler. However, it is known that the bounded Laplace distribution cannot preserve the same level of probabilistic guarantee (*i.e.,* the $\epsilon$-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 the loss [79, 80]. Based on this condition and Theorem 4.1, I 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 $\tau_i$, the bounded inter-arrival time

Laplace randomized mechanism is defined as

$$\widetilde{\mathcal{R}}(\tau_i, j) = \widetilde{L}(\overbrace{\eta_i(j)}^{j^{th}\text{ inter-arrival time of }\tau_i}, \overbrace{\tfrac{2J_i\Delta\eta_i}{\epsilon_i}}^{\text{scale of the noise distribution}}, \underbrace{T_i^{\perp}, T_i^{\top}}_{\text{bounds for randomized inter-arrival time}}) \tag{4.17}$$

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 $\mathrm{Lap}(\eta_i(j), \frac{2J_i\Delta\eta_i}{\epsilon_i})$ in which the distribution is centered at the target inter-arrival time $\eta_i(j)$.

The variables $T_{\perp}$, $T_{\top}$, $\Delta\eta_i$, $J_i$ and $\epsilon_i$ are extended task parameters of $\tau_i$ to be formalized in Section 4.4.7. Following Theorem 4.1, the bounded inter-arrival time Laplace randomized mechanism $\widetilde{\mathcal{R}}(\tau_i, j)$ is $\epsilon$-indistinguishable up to $J$ job instances.

### 4.4.7 Extended Real-Time System Model

With the components elaborated in previous sections, I now introduce the proposed real-time scheduler, $\epsilon$-Scheduler, that uses $\widetilde{\mathcal{R}}(\cdot)$ for randomizing inter-arrival times. In this section I first introduce an extended RTS task model that supports $\epsilon$-Scheduler, followed by discussion for how the extended task parameters should be determined for a given system to achieve job/task-level indistinguishability.

The basic RTS task model presented in Section 4.4.1 is extended to include parameters necessary for $\epsilon$-Scheduler to achieve desired indistinguishability. In $\epsilon$-Scheduler, a task $\tau_i$ 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, $\epsilon$-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 arrival time point from the drawn, randomized inter-arrival time (which also determines the deadline of the current job since I assume a implicit deadline model).

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 given based on the system's dynamics. The additional parameters $\Delta\eta_i$, $J_i$ and $\epsilon_i$ 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. I discuss the considerations for determining their values next.

### 4.4.8 Determining Inter-Arrival Time Sensitivity $\Delta\eta_i$

$\Delta\eta_i$ represents the degree of random noise needed to make two inter-arrival times indistinguishable and can be determined based on Definition 4.3. The value of $\Delta\eta_i$ should be fixed throughout the run-time once assigned. In the case that the job-level indistinguishability is to achieve for a given task $\tau_i$, the value of $\Delta\eta_i$ is determined solely by the task's period set $\mathcal{T}_i$. In this case, each task's sensitivity is independently considered can have different values. On the other hand, the task-level indistinguishability requires that the sensitivity reflects the period range of all tasks in the task set. 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 the task-level sensitivity will be greater than the 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, if either indistinguishability should be achieved.

### 4.4.9 Calculating Protection Duration $J_i$

With $\widetilde{\mathcal{R}}(\cdot)$, $\epsilon$-Scheduler is able to preserve $\epsilon_i$-indistinguishability up to $J_i$ job instances for a given task. As pointed out in Section 4.4.5, the more noise samples collected by an attacker the more likely the attacker is able to reconstruct the distribution and reveal the task's internal state. Therefore, $\epsilon_i$-indistinguishability can't be guaranteed for an infinite time. For this reason, $\epsilon$-Scheduler should be used with other security measures for a comprehensive protection against the scheduler side-channels.

There exist some security schemes that work well together in this context. For instance, one can integrate security tasks to perform periodic security checks to detect possible intrusion and anomalies [33]. With this scheme, the period of the security task (*i.e.,* 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 [38, 39] that enforces a reboot once a while. In such a case, the maximum time to restart 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 $\epsilon$-Scheduler offers security guarantee before the system may be compromised 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 based on the the task's period, so the value can be different across tasks. Let $\lambda$ be the protection duration in time, then

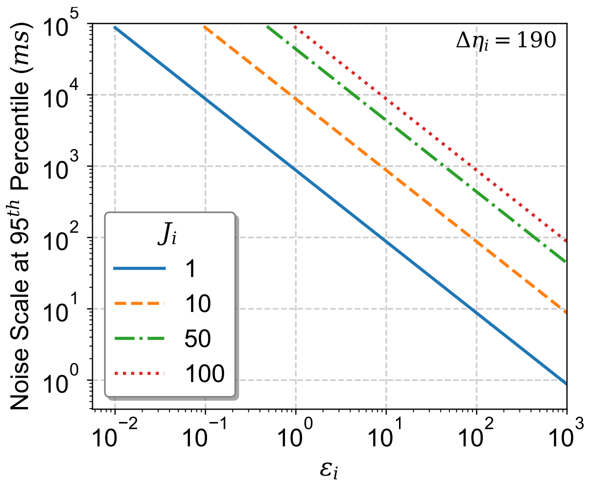


Figure 4.1: The noise scale of Lap$(0, \frac{2J_i\Delta\eta_i}{\epsilon_i})$ at $95^{th}$ percentile with $\Delta\eta_i = 190ms$ and varying $\epsilon_i$ and $J_i$. The X-axis is $\epsilon_i$ and the Y-axis is the corresponding $95^{th}$ percentile noise scale. Both axes are displayed in a base 10 logarithmic scale. The result suggests that, in the context of RTS schedule, a reasonable $\epsilon_i$ is above one order of magnitude.

$$J_i = \left\lceil \frac{\overset{\text{desired protection duration}}{\lambda}}{\underset{\text{the smallest period in the period set of } \tau_i}{\min(\mathcal{T}_i)}} \right\rceil \tag{4.18}$$

offers $\epsilon_i$-indistinguishability to $\tau_i$ within $\lambda$ time. In the case of task-level indistinguishability, $J_i$ for all tasks must be equal to offer desired indistinguishability guarantee (which is subject to $\epsilon_i$). That is,

$$J_i = \max(\left\lceil \frac{\overset{\text{desired protection duration}}{\lambda}}{\underset{\text{the smallest period in every task period set in the task set } \Gamma}{\min(\mathcal{T}_j)}} \right\rceil \mid \tau_j \in \Gamma) \tag{4.19}$$

where $\lambda$ is a global protection duration in time.

### 4.4.10 Choosing Indistinguishability Parameter $\epsilon$

With $\Delta\eta_i$ and $J_i$ determined for a given task set, $\epsilon_i$ is the major variable that a system designer specifies to secure the desired degree of protection. Ideally, a smaller $\epsilon_i$ value gives a better indistinguishability by generating randomized inter-arrival times with larger noise

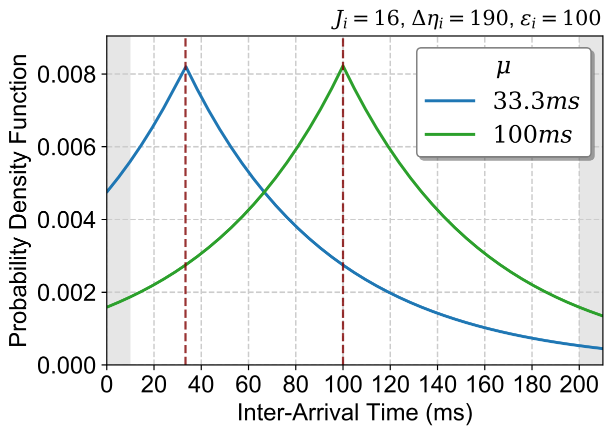


Figure 4.2: Probability density of the randomized inter-arrival times for the task $\tau_i$ with $\mathcal{T}_i = \{33.33, 100\}$. The blue and green lines show the distribution when the desired period is at $33.33ms$ and $100ms$ respectively. In this case, $\epsilon$-Scheduler offers a job-level $\epsilon$-indistinguishability for $\tau_i$ with $\epsilon_i = 100$, $\Delta\eta_i = 190$ and $J_i = 16$.

scale. However, a large noise scale may sometimes be impractical for real-time applications. Figure 4.1 shows examples of noise scales (the y-axis, represented by the $95^{th}$ percentile) with varied $\epsilon_i$ values (the x-axis) for a fixed $\Delta\eta_i = 190ms$ and various $J_i$ settings. It suggests that an $\epsilon_i$ value above an order of magnitude can be practical to most RTS. Yet, a suitable value is highly system-dependent. Ultimately, it is up to the system designer to select a best-fit value based on the overall security and performance goals. Note that all tasks must be assigned an identical $\epsilon$ value to achieve task indistinguishability while each task can have an independent $\epsilon$ value when job indistinguishability is considered.

## 4.5 IMPLEMENTATION IN REAL-TIME LINUX

I implemented both the REORDER scheduler and $\epsilon$-Scheduler in both a simulator and real-time Linux kernel running on an embedded platform. In this section I provide the platform information (summarized in Table 4.2) and a high level overview of the implementation in real-time Linux kernel.

### 4.5.1 Platform and Operating System

I used a Raspberry Pi 4 (RPi4) Model B[4] development board as the base platform for my implementation. The RPi4 is equipped with a 1.5 GHz 64-bit quad-core ARM Cortex-A72 CPU developed on top of Broadcom BCM2711 SoC (System-on-Chip). RPi4 runs on

[4]`https://www.raspberrypi.org/products/raspberry-pi-4-model-b/`.

Table 4.2: 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` |

a vendor-supported open-source operating system, *Raspbian* (a variant of Debian Linux). I forked the Raspbian kernel and modified it to implement the proposed $\epsilon$-Scheduler. Since I focus on the single core environment in this work, 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.

### 4.5.2 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* [81] 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 work, I use a PREEMPT_RT-patched kernel (4.19.71-rt24+) to enable the real-time functionality.

To further enable the fully preemptive functionality from the PREEMPT_RT patch, the `CONFIG_PREEMPT_RT_FULL` flag was enabled during the kernel compilation phase. Furthermore, the system variable `sched_rt_runtime_us` was set to $-1$ to disable the throttling of the real-time scheduler. This setting allowed the real-time tasks to use up the entire 100% CPU utilization if required[5]. Also, the active core's `scaling_governor` was set to `performance` mode to disable dynamic frequency scaling during the experiments.

[5]This change in system variable settings was mainly configured for the purpose of experimenting with the ideas of $\epsilon$-Scheduler only. For most real use-cases, users can keep this system variable untouched for more flexibility.

### 4.5.3 Vanilla EDF Scheduler

Since Linux kernel version 3.14, an EDF implementation (`SCHED_DEADLINE`) is available in the kernel code base [65]. As the PREEMPT_RT-patched kernel supports `SCHED_DEADLINE`, I used this as the baseline EDF implementation and extended the scheduler to implement $\epsilon$-Scheduler.

In Linux the system call `sched_setattr()` is invoked to configure the scheduling policy for a given process. By design, `SCHED_DEADLINE` 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.

### 4.5.4 Implementation of REORDER

```
struct sched_dl_entity {
/* task specific parameters */
  u64 dl_runtime;    // WCET
  u64 dl_deadline;   // relative deadline
  u64 dl_period;     // period
  s64 reorder_wcib;      // worst-case inversion budget

/* task instance (job) specific parameters */
  s64 runtime;       // remaining runtime
  u64 deadline;      // absolute deadline
  s64 reorder_rib;     // remaining inversion budget
  ....
/* Other variables are omitted for readability. */
};
```

Listing 4.1: REORDER parameters added to the existing data structure.

**Task/Job-specific Variables.** The Linux kernel defines a structure, `struct sched_dl_entity`, dedicated to `SCHED_DEADLINE`, to store task and job-related variables (both run-time and static variables). They include typical EDF task parameters (*e.g.,* period, deadline and WCET). To implement REORDER I added two additional variables, named `reorder_wcib` and `reorder_rib`, both `s64` (signed 64 bit integer) type variables, to store the WCIB for the task and to track the RIB for the task's active job at any given moment, respectively. Each task's `reorder_wcib` is initialized and updated when a new task is created. The job-specific run-time variable, `reorder_rib`, is initialized to the precomputed `reorder_wcib` every time when a new job arrives. During run-time, the inversion budget was updated (*i.e.,* decreased by the elapsed time in the case of priority inversion) along with other `SCHED_DEADLINE` run-time variables in the function `update_curr_dl`(). It is used to determine whether the inversion budget was consumed and a random selection of a job was allowed at a scheduling point. In my implementations I did not use any external libraries and only used the built-in kernel functions. Listing 4.1 shows a part of the existing variables as well as the newly added ones (the highlighted lines).

**Task Selection Function.** The REORDER protocol was implemented as a function, named `pick_rad_next_dl_entity`(), that selects a task and sets the next scheduling point based on the REORDER algorithm. It replaces the original `SCHED_DEADLINE` function, `pick_next_dl_entity`() (*i.e.,* one that picks the task that has the next absolute deadline from the run queue, *viz.,* the leftmost node in the scheduler's red-black tree). This function is indirectly called by the main scheduler function `__schedule`() when the next task for execution is needed.

**Randomization Function.** I used the built-in random number generator in the kernel. It supports the system call `get_random_bytes`() defined in `linux/random.h`. It is used by the function `pick_rad_next_dl_entity`() to select a random task and a random execution interval for the next scheduling point as explained in Algorithm 4.1.

**Schedule Timer.** A high-resolution timer (*i.e.,* `struct hrtimer`) was used to trigger the additional scheduling points introduced by the REORDER protocol, as described in Algorithm 4.1 (Line 22 and 23). Since this timer is a scheduler-specific timer, it is stored in `dl_rq`, as `reorder_pi_timer`. It is worth noting that `hrtimer` is also used by `SCHED_DEADLINE` to enforce the task periods.

**Idle Time Scheduling.** As introduced in Section 4.3.5, idle times are considered when the idle time scheduling scheme is deployed. In my Linux kernel implementation, I utilized the native idle task maintained under the `SCHED_IDLE` scheduler for this purpose. The REORDER protocol yields its scheduling opportunities (to other schedulers such as `SCHED_IDLE`) if $\tau_I$, the idle task in the REORDER protocol, is selected and running. The subsequent scheduling point is enforced by `reorder_pi_timer`.

### 4.5.5 Implementation of $\epsilon$-Scheduler

In my implementation I make $\epsilon$-Scheduler as a scheduling mode under `SCHED_DEADLINE` that can be enabled/disabled by setting a custom kernel parameter[6]. The $\epsilon$-Scheduler's main functionality is implemented in the function `replenish_dl_entity()` that gets invoked whenever a new job of a real-time task arrives. In this function, $\epsilon$-Scheduler generates a randomized inter-arrival time based on the Laplace distribution associated with the current task (detailed next). The generated inter-arrival time is used to compute the absolute 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.

$\epsilon$-Scheduler requires to generate 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 mathematics calculation (*e.g.,* logarithm calculation) 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 pick (with using `get_random_bytes()`) a number from the array that's associated with the desired Laplace distribution.

The conversion is done by using Algorithm 4.2. 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 4.3 as the dash curve. Line 3 computes the resolution of the percentage each point in the array represents. Line4 to line 7 iterate through each of the computed percentile to obtain and store the corresponding percent point value in the output array. Line 8 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 4.3). It is done to save memory space as a Laplace distribution is symmetric. I then use Algorithm 4.3 to obtain a random number from the PPF array.

[6]The custom kernel parameter is accessible at `/proc/sys/kernel/sched_dl_mode`.

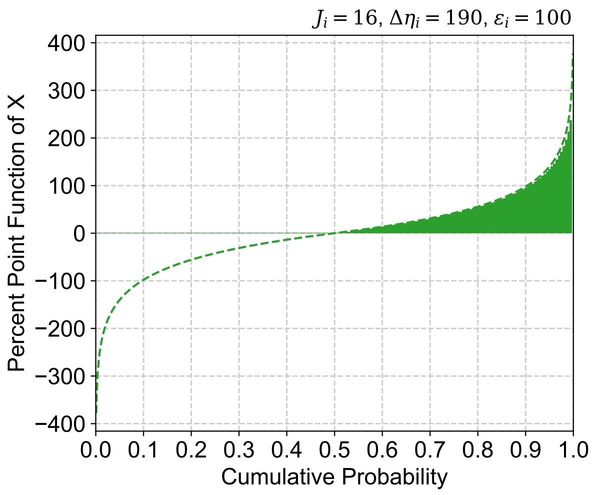


Figure 4.3: Chart of the percent of function (PPF) based on a Laplace distribution with $\epsilon_i = 100$, $\Delta\eta_i = 190$ and $J_i = 16$ (the same as that shown in Figure 4.2). The dash line represents the true PPF curve and the bars are reconstructed by the 100 PPF points stored in the PPF-based distribution array converted using Algorithm 4.2.

Algorithm 4.3 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 6 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.

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 space to store an array converted from PPF. Yet, as demonstrated by my implementation, an `u32` (*i.e.,* `unsigned int`) array storing 100 PPF points (which takes up 400 bytes space) is sufficient to produce the desired distribution. An example of the histogram for the generated random inter-arrival times drawn by the implemented $\epsilon$-Scheduler in RT Linux for a task with a target period $100ms$ is shown in Figure 4.4.

## 4.6 EVALUATION METRICS AND SETUP

The proposed REORDER and $\epsilon$-Scheduler are evaluated in a simulation platform 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 overhead in realistic environment. Furthermore, to demonstrate the applicability of $\epsilon$-Scheduler,

**Algorithm 4.2:** 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      $percentile = step \cdot resolution + 0.5$
6      $array_{PPF}[step] = int(PPF(percentile))$
7      $step = step + 1$
8  return $array_{PPF}$
```

**Algorithm 4.3:** 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      $sample = -array_{PPF}[rad_{idx} - size_{array}]$
5  else
6      $sample = array_{PPF}[rad_{idx}]$
7  return $sample$
```

additional tests are conducted on a 1/18 scale RC rover running a real application (*i.e.*, RoverBot[7], an open source autopilot system) on the RPi4 platform. In this section I first introduce metrics for evaluating $\epsilon$-Scheduler, followed by experiment setup. The evaluation results are presented in next section.

### 4.6.1 Evaluation Metrics

**Discrete Fourier Transform-Based Analysis.** Since we are concerned about the periodic components in the task schedules, frequency spectrum analysis tools such as Discrete Fourier Transform (DFT) can be useful. To adequately utilize such a tool, the task schedule

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

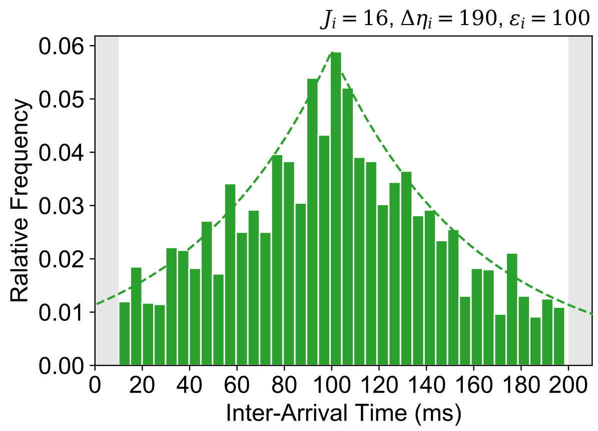


Figure 4.4: Histogram of the randomized inter-arrival times generated by $\epsilon$-Scheduler for the task $\tau_i$ with a desired period $100ms$ running in RT Linux. The extended task parameters are assigned to be $\epsilon_i = 100$, $\Delta\eta_i = 190$ and $J_i = 16$ (the same as that shown in Figure 4.2). The plot shows that the generated inter-arrival times are distributed under the desired Laplace distribution indicated by the dash line.

must be transformed into a sequence of equal-spaced samples that represent the states when CPU is busy and idle. In my analysis, a sample is taken at each time tick and hence the Nyquist frequency is half of the tick rate. Differing from the prior work [48] where busy and idle states are translated into binary values 1 and 0, I 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 by 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 4.5, the resulting frequency spectrum is useful for uncovering the degree of periodicity introduced by the scheduling of the real-time tasks. Additionally, it can also be seen that the peaked frequencies encapsulate the the true frequencies of the tasks in the demonstrative task set, annotated by the red dashed lines. It's worth noting that the spectrum can contain peaked aliasing frequencies 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.

Differing from the prior work [48] that focuses on identifying exact periods, I'm more interested in the amount of information that an adversary can learn from the DFT analysis *w.r.t.* task's periodicity. By the nature of DFT, the amplitude in the spectrum has a positive correlation with the degree of periodicity encapsulated in the sample sequences and the peaks that stand out are particularly helpful to adversaries in gaining more knowledge about the schedule.

To this end, I use a Z-score based peak detection algorithm [82, 83] 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 4.5, such a moving threshold can effectively identify the peaks that are significant while filtering out the base noise.

**Upper-Approximated Schedule Entropy and Average Slot Entropy.** The notion of *Schedule Entropy* was first introduced by Yoon *et al.* [49] to calculate the randomness given to a task schedule by the TaskShuffler scheduling algorithm. Yoon then proposed *Upper-Approximated Schedule Entropy* to empirically estimate the schedule entropy of a given task set. A bound is then derived by Vreman *et al.* [84] showing the legitimacy of such estimation. The upper-approximated schedule entropy is calculated by [49, Definition 5]

$$\widetilde{H}_\Gamma(\mathcal{S}) = \sum_{t=0}^{L-1} \overset{\text{slot entropy function}}{H_\Gamma}\,(\underset{\text{slot t in the schedule vector } \mathcal{S}}{S_t}) \tag{4.20}$$

where $\mathcal{S}$ is a $L$-dimensional random vector $\mathcal{S} = (S_0, ..., S_{L-1})$ that represents a task schedule of length $L$ and $H_\Gamma(\cdot)$ is the slot entropy calculated by

$$H_\Gamma(S_t) = -\sum_{s_t=1}^{N} \overbrace{\Pr(s_t) \log_2 \Pr(s_t)}^{\text{probability mass function of a task appearing at a time slot } t} \tag{4.21}$$

where $\Pr(s_t)$ is the probability mass function of a task appearing at a time slot $t$. As shown in Equation 4.20, the scale of the resulting entropy depends on the length of the schedule under analysis. In this work, I use *Average Slot Entropy* [84] that calculates the mean slot entropy based on the upper-approximated schedule entropy (*i.e.*, $\frac{\widetilde{H}_\Gamma(\mathcal{S})}{L}$).

**ScheduLeak Inference Precision and Inference Success Rate.** I test the developed schedulers against the introduced scheduler side-channel attack (*i.e.*, DyPS in DP RTS). Therefore the inference precision, $\mathbb{I}_v^o$ (Definition 3.5), introduced in Section 3.6.1 as well as the inference success rate are used as metrics to evaluate the effectiveness of defense.

**Quality-of-Service (QoS).** Other than the security part of the metrics, it is also important to understand the impact on the timing properties (*e.g.*, deadlines misses, periodicity of the task execution) that are vital to delivering promised services in RTS applications. What follows describes a set of metrics I use to examine the quality-of-service with respect to fulfilling the RTS timing requirements.

- **Deadline miss ratio.** A deadline miss ratio is commonly used as a measure of QoS for a real-time task. It is defined as the ratio of the number of deadline misses to the total number of completed and aborted task instances [76].

- **The number of consecutive deadline misses.** While deadline misses are generally tolerable to the RTS in our context, an unbounded number of consecutive deadline misses can disrupt the service delivery depending on the application of the real-time tasks [75].

- **Mean task frequency.** Given that a real-time task is initially designed to deliver services at a certain frequency, a biased frequency can disturb the task's ability to accomplish such a target. A task's mean frequency is calculated by the inverse of the mean inter-arrival times.

- **Under-performance ratio.** 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 than expected (*i.e.,* with inter-arrival times larger than the desired period) as a under-performing execution usually has direct impact on the task's commitment to the service delivery. I 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.

### 4.6.2 Evaluation Setup

A synthetic task set with timing parameters of an avionic system from prior work [32] that has a task set utilization 0.64 is used to specifically examine the outcome of the proposed scheduler in the first part of the evaluation. The task set parameters are shown in Table 4.3.

The proposed scheduler is also tested using extensive synthetic task sets that are generated from a generation mechanism similar to that in earlier research [20, 33, 49, 85]. 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 subgroup. The utilization of each individual task in a task set is generated from a uniform distribution by using the *UUniFast*

Table 4.3: Timing Parameters of a Avionics Demonstrator [32]

| 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 |

algorithm [86]. 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 $[0, T_i)$.

To explore the best-case protection as well as the most impact on the system performance, I configure the extended task parameters based on the requirements for achieving the task-level indistinguishability. The efficacy of the job-level indistinguishability is specifically examined against the ScheduLeak attack in which a specific victim task is targeted and hence constituting an ideal scenario for testing the job-level indistinguishability (results presented in Section 4.7.3). To achieve the task-level indistinguishability in the experiments, $\Delta\eta$ is assigned to be $200ms - 10ms = 190ms$. $J_i$ for each task is calculated by using Equation 4.19 with a protection duration of $500ms$ that has been demonstrated to be practical to perform periodic security checks to provide consistent and effective protection in RTS [33][8]. I consider two $\epsilon$ settings 10 and $10^3$ which represent two end values that one may reasonably choose based on the noise range shown in Figure 4.1. In the experiments, I use a fixed simulation duration $5000ms$ so that I'm able to compare the experiment results across different task sets. Besides the proposed REORDER and $\epsilon$-Scheduler, I also include the vanilla EDF scheduler for comparison in the evaluation.

To evaluate the scheduling overhead, I conduct experiments on the RPi4 platform running RT Linux. I use the built-in `SCHED_DEADLINE` scheduler as the Vanilla EDF scheduler and a custom implementation of TaskShuffler EDF for comparison. The time cost of a function is measured by using the `trace-cmd` command. For evaluating the power consumption, I use a High Voltage (HV) Power Monitor manufactured by Monsoon[9] to supply a $5.2V$ power to the RPi4 board. The power consumption is then monitored in the power monitor's software, PowerTool version 5.0.0.25.

To demonstrate the usability of the proposed $\epsilon$-Scheduler, I conduct experiments on a 1/18

[8]It is shown that the security tasks are typically assigned periods in the range $[250ms, 500ms]$ [33]. In the evaluation, I take $500ms$ (*i.e.,* the worst protection) to estimate protection duration $J$.

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

scale rover running an open source autopilot application, RoverBot[10], on the RPi4 platform. RoverBot consists of 7 tasks (*i.e.,* Actuator, RCInput, BatteryMonitor, AHRS, Localizer, Navigator). 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[11] tracking camera that enables precise indoor localization as well as indoor navigation. Such features allow me to study and measure the degree of the performance degradation caused by $\epsilon$-Scheduler in a real application. The experiment results are presented in Section 4.7.6.

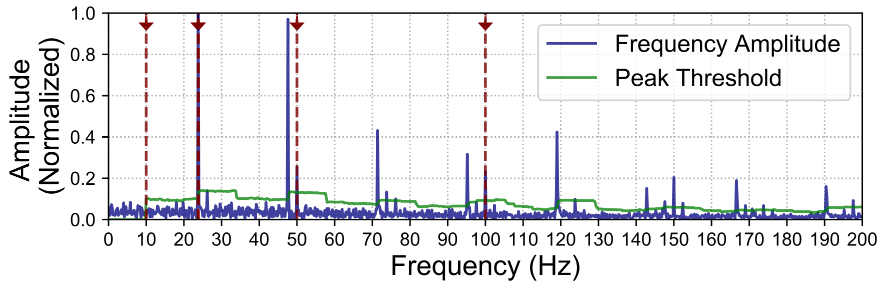
## 4.7 EVALUATION RESULTS

### 4.7.1 DFT-Based Analysis

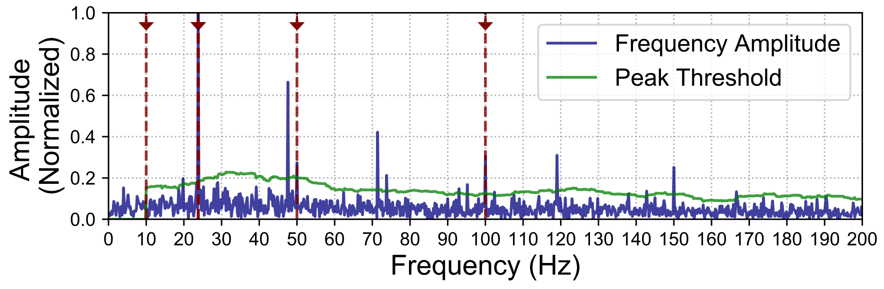
First I set off to understand the periodicity enclosed in the schedules produced by the Vanilla EDF scheduler, the REORDER scheduler and $\epsilon$-Scheduler (with $\epsilon = 10^3$ and $\epsilon = 10$). I do this by analyzing the DFT of the schedules based on the task set introduced in Table 4.3 and the resulting frequency spectra are shown in Figure 4.5. As revealed by the peaks displayed in Figure 4.5(a), the task periods are enclosed in the schedule generated by the vanilla EDF scheduler because of its work-conserving nature. It's worth pointing out that the period $100ms$ (*i.e.,* $10Hz$) 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 4.5(b) shows the spectrum of the same task set scheduled under the REORDER scheduler and the result is similar to that under the vanilla EDF scheduler with more base noise. This is due to the high task set utilization (*i.e.,* 0.64 in this case) that gives a smaller chance of schedule obfuscation. While the task set may not represent every case, it does demonstrate the shortcoming of the TaskShuffler's randomization protocol – it gets less effective when the task set utilization is high. This shortcoming can also be seen in later experiments. On the other hand, Figure 4.5(c) and (d) show the spectra when scheduled under $\epsilon$-Scheduler with $\epsilon = 10^3$ and $\epsilon = 10$, respectively. Both settings create huge noise across the entire frequency domain. As a result, no peaks match the task frequencies significantly.

The green lines shown in Figure 4.5 are the moving peak threshold calculated by using the Z-score based peak detection algorithm introduced in Section 4.6.1. From the figures we can see that the threshold is useful for identifying the outstanding peaks while filtering out the background noise. The outstanding peaks represent the true periodicity coming out of the schedule and thus are particularly useful for attackers to reconstruct and leak the

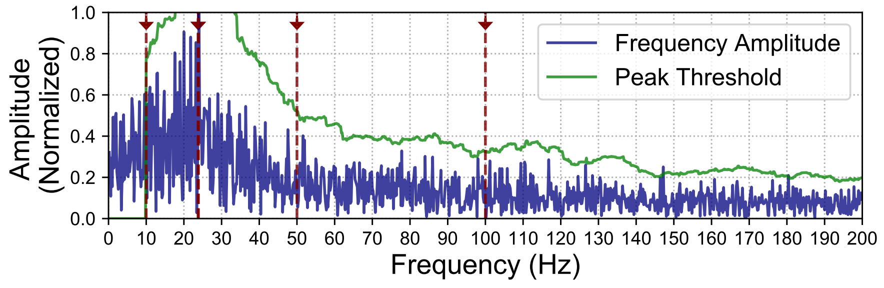
[10]`https://github.com/bo-rc/Rover`
[11]`https://www.intelrealsense.com/tracking-camera-t265`



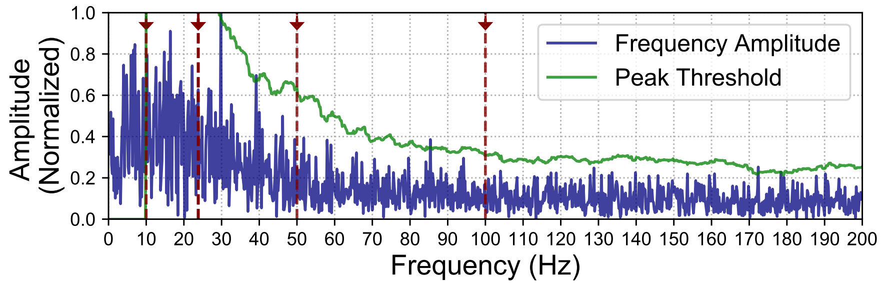
(a) Vanilla EDF



(b) REORDER



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



(d) $\epsilon$-Scheduler ($\epsilon = 10$)

Figure 4.5: Results of the DFT-based analysis over the demonstrative task set given in Table 4.3 scheduled by the Vanilla EDF, REORDER and $\epsilon$-Scheduler schedulers. The blue lines are the normalized amplitudes for the corresponding frequency bins and the green lines are the Z-scored based moving peak threshold (introduced in Section 4.7.1) for detecting outstanding peaks. The results suggest that $\epsilon$-Scheduler creates a wide range of noise in the frequency spectrum and is effective in obscuring the periodic elements enclosed in the original schedule.

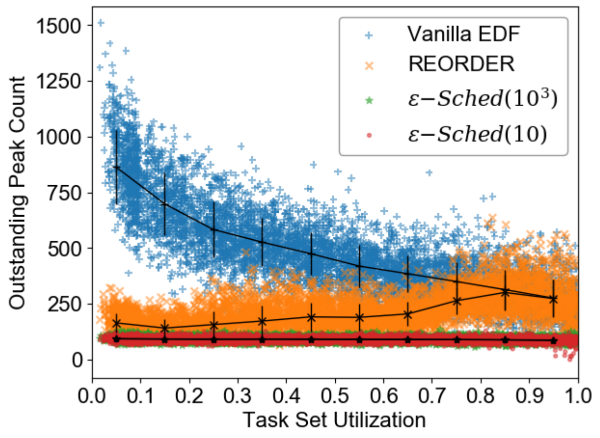


Figure 4.6: Results of the detected outstanding peak count. The results are grouped by the task set utilization (x-axis) and the y-axis is the outstanding peak count. Each point is the result of a task set scheduled by the corresponding scheduler. It indicates that Vanilla EDF yields a large number of peaks that are useful for adversaries to learn the schedule while there are no significant amount of peaks detected with $\epsilon$-Scheduler.

task set information. Intuitively, the more outstanding peaks the attackers collect, the more precise information the attackers can evaluate and learn. Next I use the aforementioned peak detection algorithm to count the number of outstanding peaks and test with extensive synthetic task sets to get a broader understanding of the effectiveness of $\epsilon$-Scheduler over obscuring the task periodicity. The experiment results are plotted in Figure 4.6 where each point represents the result of a task set for the corresponding scheduler. As expected, the vanilla EDF scheduler yields schedules with stronger periodicity and more outstanding peaks. On the other hand, the REORDER scheduler can effectively obscure the task periodicity for most of the task sets except those with higher utilization. With $\epsilon$-Scheduler, no significant amount of outstanding peaks is detected due to larger overall noise in both $\epsilon = 10^3$ and $\epsilon = 10$ settings. The result also indicates that the efficacy of $\epsilon$-Scheduler, differing from REORDER, is independent to the task utilization.

### 4.7.2 Average Slot Entropy

Next I analyze the schedules by measuring their average slot entropy introduced in Section 4.6.1. The results are shown in Figure 4.7. Similar to Figure 4.6, a point represents the average slot entropy of a task set scheduled under the corresponding scheduler. The results indicate that $\epsilon$-Scheduler yields higher entropy than the other two schedulers even when the task set utilization is high in which REORDER fails to obfuscate the schedules. In $\epsilon$-Scheduler, $\epsilon = 10$ generally performs better than $\epsilon = 10^3$ *w.r.t.* the entropy as the former

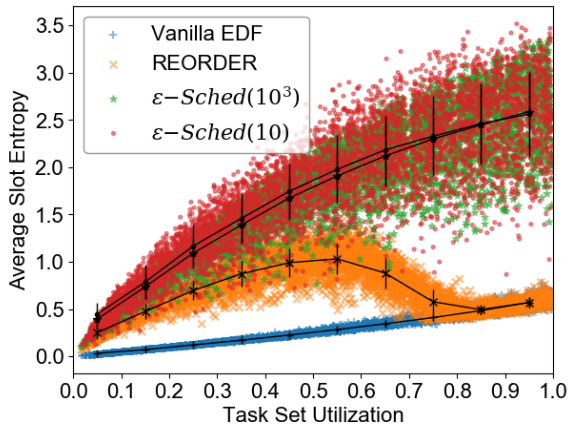


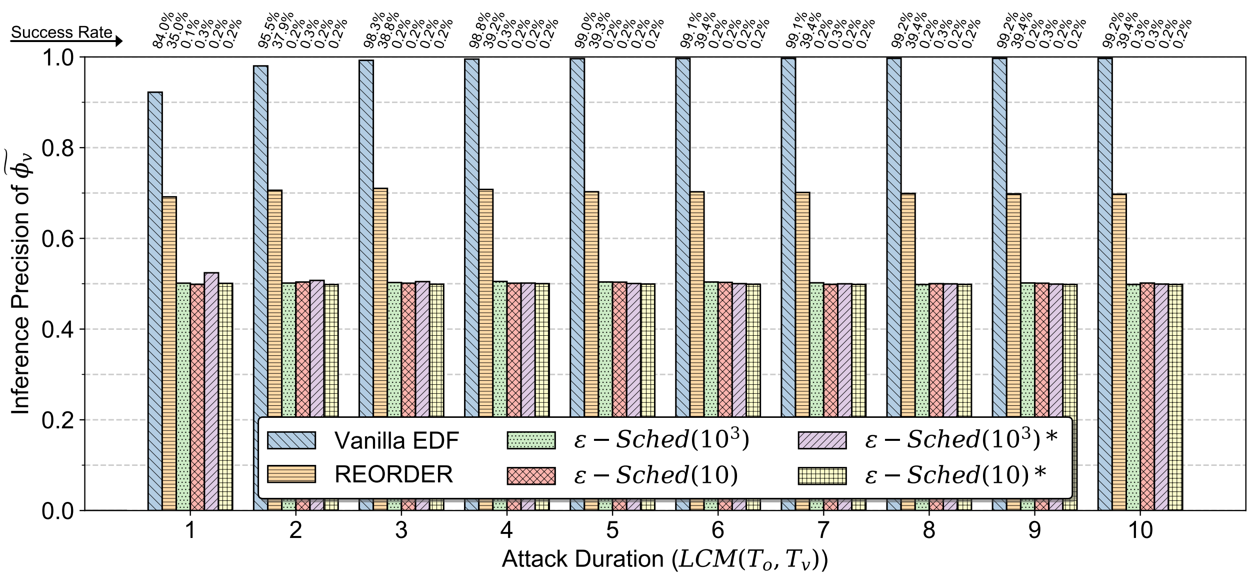
Figure 4.7: Results of the average slot entropy (y-axis) grouped by the task set utilization (x-axis). Each point is the result of a task set scheduled by the corresponding scheduler. It shows that $\epsilon$-Scheduler generates diversified schedules with higher entropy (*i.e.,* more randomness).

has a wider variation range for the randomized inter-arrival times.

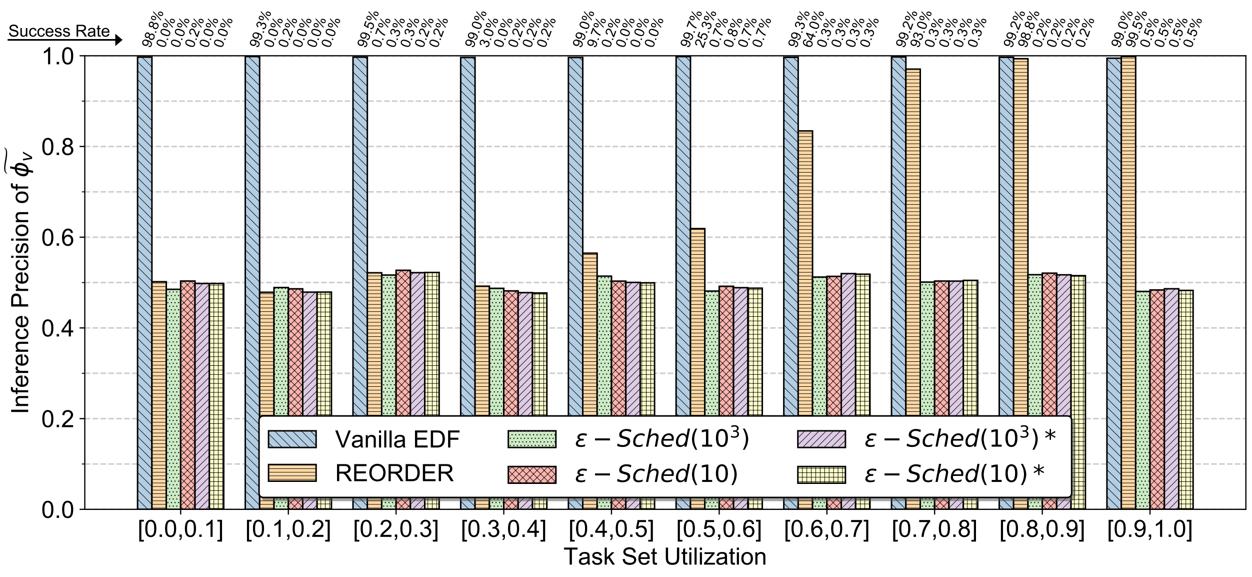
### 4.7.3 ScheduLeak Inference Precision and Inference Success Rate

To understand the effectiveness of the schedulers against the scheduler side-channel attacks, I perform the ScheduLeak attacks over the generated synthetic task sets. In this experiment, the observer task and the victim task in a task set are selected from the generated tasks based on their task periods. 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 task is then selected as 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 $T_v$.)

I first run experiments with configurations for achieving task indistinguishability. The experiment results are shown in Figure 4.8(a) and (b). As indicated in Figure 4.8(a), while ScheduLeak gains better inference precision and success rate as attack duration increases in the case of vanilla EDF and REORDER, $\epsilon$-Scheduler offers consistent protection throughout the entire course of the attack. Figure 4.8(b) shows the breakdown of the inference results grouped by the task set utilization at the attack duration $10 \cdot LCM(T_o, T_v)$. It reveals that the REORDER scheduler offers less effective protection when the task set utilization is high due to limited randomization restricted by the strict real-time requirements. On the other hand, $\epsilon$-Scheduler yields consistent performance across the task utilization leading to an



(a) Inference results with varying attack duration



(b) Inference results grouped by the task set utilization

Figure 4.8: Results of the inference precision and success rate produced by the ScheduLeak attack against various scheduler settings. The figure (a) shows the inference results for an attack duration ranged from $1 \cdot LCM(T_o, T_v)$ to $10 \cdot LCM(T_o, T_v)$ and the figure (b) shows the inference results of $10 \cdot LCM(T_o, T_v)$ grouped by the task set utilization. The inference success rate for each group is displayed at the top of the figures. The experiment suggests that $\epsilon$-Scheduler can offer effective protection against the ScheduLeak attack. Such an effect is independent to the attack duration and the task set utilization.

average inference precision (0.498 and 0.501 for $\epsilon = 10^3$ and $\epsilon = 10$) that is close to the outcome produced by a random guess.

Next, I test if configuring only the victim task to achieve its job indistinguishability is sufficient to protect it against the ScheduLeak attack. In this experiment, all tasks have consistent inter-arrival times scheduled based on their periods (*i.e.,* $\epsilon_i = \infty$) except the victim task. The results are presented as the $5^{th}$ ($\epsilon$-$Sched(10^3)^*$) and $6^{th}$ ($\epsilon$-$Sched(10)^*$) bars in each group shown in Figure 4.8(a) and (b). As shown, the victim task is protected with the job indistinguishability. The ScheduLeak attack fails to take advantage of the scheduler side-channels and yields the inference precision at a level similar to a random guess. The inference success rate results displayed at the top of the figures reveal the same trend, showing that the attacks have either zero or a very small chance to succeed (*i.e.,* compute a correct inference) when the $\epsilon$-Scheduler is adopted.

### 4.7.4 QoS-Based Analysis

While the above results show that the proposed $\epsilon$-Scheduler is effective in creating noise in the schedule, I'm interested in learning the impact on the QoS of the tasks. I first examine the case of deadline misses in the experiments. As expected, both Vanilla EDF and REORDER obey strict real-time constraints and thus do not yield any deadline misses. In $\epsilon$-Scheduler, no deadline miss has been observed when $\epsilon = 10^3$. However, in the case of $\epsilon = 10$, I observe intermittent deadline misses in some of the task sets with high utilization. The number of task sets that have encountered deadline misses in such a setting is plotted in the top section of Figure 4.9. 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.

I next examine how close to the tasks performing to the desired execution frequencies. In this experiment, a task's frequency error is calculated by 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 4.9. The result indicates that the task sets scheduled by $\epsilon$-Scheduler with $\epsilon = 10$ has frequency error significantly larger than that with $\epsilon = 10^3$. It is expected as $\epsilon = 10$ yields wider inter-arrival time range. It is also worth pointing out that the frequency error is due to the bounding in generating the randomized inter-arrival times which can lead to a asymmetric distribution (as an example, see the distribution for $\mu = 33.3ms$ in Figure 4.2). Similar result can also be observed in the measurement for the under-performance ratio. In this experiment, I first obtain the worst under-performance ratio for each task set (by measuring the under-performance ratio of each task and take the worst in the task set) and then calculate the mean of the worst under-performance ratio. This allows us to understand the worst under-performance ratio a system may observe from its tasks. As shown in Figure 4.10, the under-performance ratio

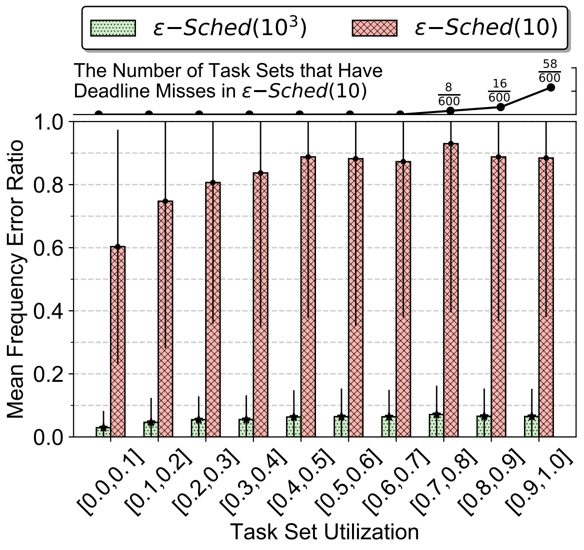


Figure 4.9: Results of the measurement for the mean frequency error ratio (x-axis) grouped by the task set utilization (y-axis). It shows that a large $\epsilon$ value can lead to greater mean frequency error and also cause some tasks to miss deadlines when the task set utilization is high, as displayed by the plot at the top section that shows the number of task sets that have experienced deadline misses in each utilization group with $\epsilon = 10$.

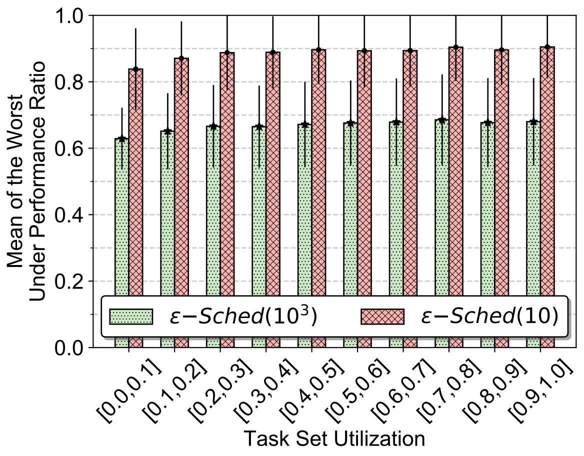


Figure 4.10: Results of the mean of the worst under-performance ratio (y-axis) grouped by the task set utilization (x-axis). The experiment gives us an insight into the degradation a system may observe from its tasks. It suggests that a task's under-performance ratio can be biased towards 0.5 and above. The bias is noticeable when $\epsilon$ is large. This often happens on the task that has a small period leading to a asymmetric distribution that tends to generate larger inter-arrival times.

can be biased towards 0.5 and above, leading to a degradation in the execution frequency. This usually happens on the task that has a small target period and hence a asymmetric distribution that tends to generate larger inter-arrival times (again, see Figure 4.2 for an

Table 4.4: Summary of Scheduling Overhead Measurement

| | Vanilla EDF | REORDER | $\epsilon = 10^3$ | $\epsilon = 10$ |
|---|---|---|---|---|
| Mean Context Switch Count Ratio | 1 | 2.525 | 0.914 | 0.696 |
| Mean `pick_next_dl_entity()` Cost | $1.25us$ | $4.3us$ | $1.44us$ | $1.39us$ |
| Mean `get_next_inter_arrival_time()` Cost | - | - | $5.79us$ | $5.41us$ |
| Average Power Consumption (`performance`) | $2371.78mW$ | $2389.1mW$ | $2377.12mW$ | $2360.2mW$ |
| Average Power Consumption (`ondemand`) | $2198.04mW$ | $2303.77mW$ | $2075.8mW$ | $2045.3mW$ |

example). It hints that one should expect a degradation in the service (with respect to the task frequency) when using $\epsilon$-Scheduler, particularly with a small $\epsilon$ value (*i.e.,* larger noise and variation in the schedule).

### 4.7.5 Scheduling Overhead

I next evaluate the scheduling overhead of the proposed $\epsilon$-Scheduler, together with the Vanilla EDF and REORDER schedulers as a comparison. The measurement results are summarized in Table 4.4.

First, I take Vanilla EDF as the base and calculate the context switch count ratio compared to REORDER and $\epsilon$-Scheduler in simulation. The result suggests that REORDER 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, $\epsilon$-Scheduler produces fewer context switches as the generated inter-arrival times can be greater (*i.e.,* task executing less frequently). This measurement generally matches the result shown in Figure 4.10.

Next, I run a synthetic task set with task parameters given in Table 4.3 in RT Linux on the RPi4 platform to measure the mean cost of the scheduling. I first measure the time cost of the main scheduling function `pick_next_dl_entity()` in which next task is being picked at a scheduling point. The result shows that REORDER has larger overhead as it invokes `get_random_bytes()` which takes an average $2.23us$ to generate a 64-bit random number to shuffle the task selection. On the other hand, $\epsilon$-Scheduler has overhead similar to Vanilla EDF as the scheduling mechanism is identical in `pick_next_dl_entity()`. To evaluate true overhead of $\epsilon$-Scheduler, I measure the function `get_next_inter_arrival_time()` where randomized inter-arrival times are generated in my $\epsilon$-Scheduler implementation. As shown in the table, the time cost is around $5.79us$ and is independent to the $\epsilon$ 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

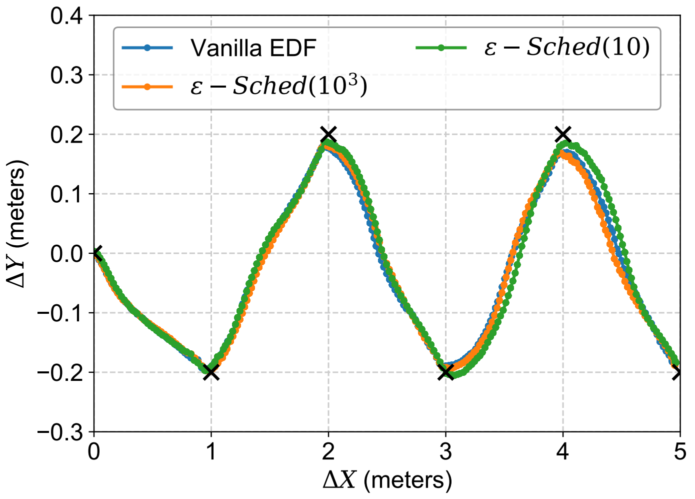


Figure 4.11: The trajectory of the rover system steering through predefined way points (the black cross markers) with RoverBot running under Vanilla EDF and $\epsilon$-Scheduler. The result indicates that larger diversification and higher protection under $\epsilon$-Scheduler with $\epsilon = 10$ can result in larger offsets in trajectory. The worst observed deviations are $0.029m$ and $0.041m$ 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.

to the context switch overhead as an arrival of a job in EDF (and $\epsilon$-Scheduler) does not necessarily lead to a re-scheduling (*i.e.,* a `pick_next_dl_entity()` call).

I then measure the power consumption of the platform running the task set with each of the schedulers. When the scaling governor is configured as `scaling_governor` = `performance` (which is a typical setting for RTS to maintain a predictable execution time and behavior), the power consumption consistent for all schedulers. It is expected as CPU runs at the top frequency at all times under the `performance` setting. As a comparison, I measure the power consumption with `scaling_governor` = `ondemand` which 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 $\epsilon$-Scheduler) and suggest that $\epsilon$-Scheduler does not result in higher power consumption.

### 4.7.6 Evaluation on a Real Application

This part of the evaluation studies the impact of $\epsilon$-Scheduler on real applications by running the RoverBot autopilot software on the RPi4 platform with $\epsilon$-Scheduler enabled on a 1/18 scale rover system. To better analyze the performance variation, I focus on diversifying a single task, 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. I let the rover steer through a series of predefined way points (in this case, a wavy route) and record the resulting trajectory under Vanilla EDF and $\epsilon$-Scheduler with $\epsilon = 10$ and $\epsilon = 10^3$. The experiment results are shown in Figure 4.11 in which the predefined way points are displayed as black cross markers starting 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 becomes $65.06Hz$ with $\epsilon = 10^3$ and $10.22Hz$ with $\epsilon = 10$. Taking the trajectory of Vanilla EDF as the ground truth, the largest recorded deviations are $0.029m$ and $0.041m$ for $\epsilon = 10^3$ and $\epsilon = 10$ respectively. This result has the same implication as what we have learned from the previous experiments in simulation – larger task diversification (and hence higher protection) results in increased performance degradation. On the other hand, the trajectories show that the rover is able to reach 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 $\epsilon$-Scheduler can be applied to real applications and also meet users needs (*e.g.,* better protection or better performance) using the adjustable $\epsilon$ parameter.

## 4.8 DISCUSSION

From the presented evaluation, both $\epsilon = 10^3$ and $\epsilon = 10$ settings in $\epsilon$-Scheduler produce promising results for obscuring the periodicity and diversifying the schedule. However, as shown by the QoS measurement, the $\epsilon = 10^3$ setting yields a more reasonable variation in the task frequency range. As a result, there is a trade-off between the degree of protection and the determinism in RTS when using $\epsilon$-Scheduler. While $\epsilon$-Scheduler offers less analytical protection with $\epsilon = 10^3$ value, it may not be unusual to choose such a large $\epsilon$ value in most RTS due to the limitations in the tolerable frequency changes. A possible improvement is to dynamically adjust the $\epsilon$ value based on the QoS and protection demand at run-time. In such a case, the $\epsilon$ value is particularly useful as a security parameter to be integrated with a feedback control real-time scheduling algorithm (*e.g.,* [76]).

Another issue resides in the interpretation of the chosen indistinguishability parameter $\epsilon$ value. While the evaluation shows that the proposed $\epsilon$-Scheduler can satisfy the user's needs by adjusting the indistinguishability parameter $\epsilon$, it is in fact unrealistic to derive a meaningful guarantee for an absolute $\epsilon$ value as the same assignment may yield very different protections on different systems. Similar to differential privacy, it is more meaningful to compare the relative performance between two values for a given system like what I demonstrated in the evaluation. As my experiment results have shown, there exist many metrics that offer more insights into the performance degradation as well as the degree of protection against certain attacks. Therefore, it is more reasonable to directly examine the measurement from the metrics for the properties and protection guarantees that users care about while determining a proper $\epsilon$ value.

With the presented evaluation results for $\epsilon$-Scheduler, I believe that the schedule indistinguishability concept can be extended to other system properties (*e.g.,* memory, code, data) as well to further diversify the system behavior from many more aspects. The outcome is the possibilities to improve the security of a broader class of systems, *e.g.,* distributed IoT and general-purpose operating systems.

## 4.9 CONCLUSION

Malicious attacks on systems with safety-critical real-time requirements could be catastrophic since the attackers can destabilize the system by inferring the critical task execution patterns. In this work I explore a scheduling mechanism for obscuring the embedded periodicity and diversifying the task schedule. With using proposed scheduler, the system designer now has the ability to protect the system with analytical guarantee. As a result, this validates my hypothesis and answers the second key research question I raised in Section 1.1 with respect to the defense techniques – diversifying the real-time schedule is effective in defending against the scheduler side-channels in preemptive RTS.

# CHAPTER 5: ANALYTIC AND EVALUATION FRAMEWORK

As the scheduler side-channels exist in any RTS with deterministic behaviors, the impact can be broad. Therefore, a comprehensive measurement and checking mechanism to analyze the risks of the scheduler side-channels for a given system is in demand. In this chapter, I introduce an analytic and evaluation framework that others can follow systematically. To this end, I propose a 3-dimensional metric system to evaluate a given system. The three dimensions are: *(i)* degree of indistinguishability, *(ii)* degree of protection against attacks and *(iii)* quality-of-service (QoS). I elaborate the definition and examples for each dimension next. Figure 5.1 visualizes this 3-dimensional metric system and some exemplified scenarios are illustrated in Section 5.4.

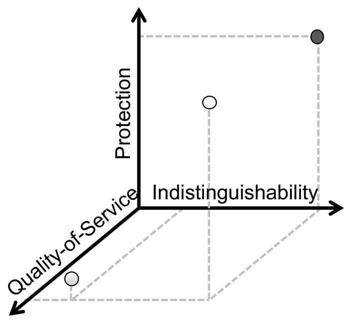


Figure 5.1: A 3-dimensional metric system for evaluating the degree of indistinguishability, the degree of protection and quality-of-service of the given system. The white, grey and black points exemplify three of the many possible results that represent the best, an undesired and a typical scenarios respectively.

## 5.1 DEGREE OF INDISTINGUISHABILITY

The first dimension aims to evaluate the indistinguishability offered by the system under a certain setting. While the desired indistinguishability is achieved by designing the system to follow the strict theoretical bound and distribution model, a checking mechanism that measures such indistinguishability guarantee from outside of the system core is needed to learn the true effect of the protection.

As the indistinguishability guarantee comes from the carefully designed noise distribution, we may reconstruct the *noise distribution* to verify the correctness of the system behavior with respect to the indistinguishability. As shown in Figure 4.4 demonstrated by the measurement on $\epsilon$-Scheduler running in RT Linux, the distribution of the generated noise can be reconstructed to examine if the design is correct and the system is behaving as expected.

Another way to evaluate the indistinguishability, in the case of schedule indistinguishability, is by conducting *Discrete Fourier Transform-based analysis* [48]. As introduced in Section 4.6.1, the task schedule can be transformed into a sequence of 1.0 and −1.0 samples and analyzed by using DFT analysis. The resulting frequency spectrum is a good visualization of what is actually happening in the task schedule. Then, additional computation such as the Z-score based peak detection algorithm [82, 83] that counts the number of outstanding peaks in the spectrum can be adopted to extract useful information for further analysis of the risks.

## 5.2 DEGREE OF PROTECTION AGAINST ATTACKS

In the second dimension, the framework measures the degree of protection against existing attacks. It gives the system designer an insight into how vulnerable the system is under the current setting. As an example, the *inference precision*, denoted by $\mathbb{I}_v^o$, introduced in the ScheduLeak work in Section 3.6.1 is useful to estimate the attacker's ability to leak critical information of a target task $\tau_v$ from the scheduler side-channels. It represents the precision of the inferred phase $\widetilde{\phi_v}$ compared to the true phase of a target task $\phi_v$. A larger $\mathbb{I}_v^o$ indicates that the inference $\widetilde{\phi_v}$ is more precise in inferring $\phi_v$. To measure the degree of protection based on the inference precision, we can take its complement $1 - \mathbb{I}_v^o$ which gives a positive correlation with the protection.

## 5.3 QUALITY-OF-SERVICE (QOS)

Besides the security dimensions, QoS is a dimension that focuses on the evaluating the performance degradation that's incurred by any defense technique. The metrics for QoS may vary system to system depending on the system dynamics and applications. Section 4.6.1 introduces some timing metrics (*e.g.,* deadline miss ratio, consecutive deadline misses, mean task frequency, under-performance ratio) that are essential to most RTS.

## 5.4 USAGE OF THE FRAMEWORK

The 3-dimensional metric system is useful to help system designers understand how a given system is performing with respect to the indistinguishability, protection and quality-of-service. Three scenarios are illustrated by the grey, white and black points shown in Figure 5.1. The grey point represents a case that a typical RTS, without any protection mechanisms, may perform. In such a case, QoS is guaranteed by following the strict timing requirements. The level of indistinguishability and protection is low as no additional effort is made to harden the system. On the other hand, the black point demonstrates a case where the system designer focuses solely on achieving the indistinguishability. While a good protection against attacks may be gained, it gives a bad QoS which can result in unexpected system behaviors and lead to severe consequences. The white point shows the best case where all three dimensions reach good levels. It means that the system, while offering good QoS, is mostly indistinguishable for the components under examination and is well protected.

Taking the experiment results presented in Section 4.6.2 for REORDER and $\epsilon$-Scheduler as an example, the shortcomings for each of the scheduler settings are becoming clear. To REORDER, while it performs very well in the dimension of QoS (as it obeys the real-time constraints by design), it has limited performance in the protection and indistinguishability dimensions. To $\epsilon$-Scheduler, both $\epsilon = 10$ and $\epsilon = 10^3$ settings perform exceptionally in the protection and indistinguishability dimensions. Yet, $\epsilon = 10$ results in a wider range of task frequency variation (due to its larger noise range being added into the task's inter-arrival times) that reduces the system's stability and hence has a bad QoS performance. On the other hand, while degradation can be observed in $\epsilon = 10^3$, its scale is arguably more acceptable.

The above 3-dimensional metrics resolve the third key research question raised in Section 1.1 with respect to the evaluation for the risk posed by scheduler side-channels and the efficacy of defense schemes. With such a framework, a system designer can better determine which design or configurations can meet desired performance and protection goals. In the above cases, a designer who wants to have a better QoS and mild protection against the scheduler side-channels can choose to adopt the REORDER scheme, while one that needs exceptional protection can employ $\epsilon$-Scheduler and adjust the $\epsilon$ value accordingly.

# CHAPTER 6: CONCLUSION

In this thesis, I investigate the problem of scheduler side-channels in preemptive RTS. To comprehensively understand the risks and countermeasures, I raised three key research questions (Section 1.1) and conducted research from both attacker's and defender's perspectives.

Setting off as an adversary, I demonstrated the existence of scheduler side-channels in both fixed-priority and dynamic-priority RTS in Chapter 3. In Section 3.3, I proposed the ScheduLeak attack algorithms that leak critical timing information of a high-priority real-time task by using a low-priority, unprivileged user-space task in fixed-priority RTS. In Section 3.4, I extended ScheduLeak and introduced the DyPS attack algorithms that target dynamic-priority RTS. The evaluation presented in Section 3.7 showed that the proposed scheduler side-channel attacks can infer the timing information (*i.e.,* the victim task's phase) with a high precision and the inference is particularly useful for helping later attacks achieve their attack goals better. This part of the work answers the first key research question regarding the presence of the scheduler side-channels in preemptive RTS.

Moving to the defender's side, I introduced two scheduler-based defense schemes in Chapter 4. In Section 4.3, I studied the schedule randomization technique by introducing the REORDER scheduler in dynamic-priority RTS. Limitations of such a technique caused by the strict real-time constraints are identified. In Section 4.4 I defined the notion of schedule indistinguishability and introduce $\epsilon$-Scheduler that relaxes the real-time constraints to offer guaranteed protection against the scheduler side-channels by achieving task/job indistinguishability. The evaluation presented in Section 4.6.2 showed a promising result and resolves the second key research question with respect to diversifying the schedule and defending against the scheduler side-channel attacks (*i.e.,* DyPS).

Based on the outcomes from Chapter 3 and 4, I proposed a analytic and evaluation framework that's dedicated to assessing the risks of a given system with respect to the scheduler side-channels in Chapter 5. With such a framework, a system designer now has a systematic way to assess how the system performs against the scheduler side-channels and how the system should be adjusted to meet the desired protection and performance goal – this resolves the third key research question.

Finally, with the research and experiment results presented in this dissertation, the hypothesis – *there exist scheduler side-channels in preemptive RTS that can be defended against by diversifying the real-time schedules*, can be validated.

# REFERENCES

[1] T. M. Chen and S. Abu-Nimeh, "Lessons from stuxnet," *Computer*, vol. 44, no. 4, pp. 91–93, Apr. 2011.

[2] D. U. Case, "Analysis of the cyber attack on the ukrainian power grid," *Electricity Information Sharing and Analysis Center (E-ISAC)*, 2016.

[3] D. Schneider, "Jeep Hacking 101," *IEEE Spectrum*, Aug 2015, http://spectrum.ieee.org/cars-that-think/transportation/systems/jeep-hacking-101.

[4] B. Min and V. Varadharajan, "Design and analysis of security attacks against critical smart grid infrastructures," *2014 19th International Conference on Engineering of Complex Computer Systems*, vol. 0, pp. 59–68, 2014.

[5] M.-K. Yoon, B. Liu, N. Hovakimyan, and L. Sha, "Virtualdrone: virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems," in *Proceedings of the 8th International Conference on Cyber-Physical Systems.* ACM, 2017, pp. 143–154.

[6] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *Security and Privacy (SP), 2010 IEEE Symposium on*, may 2010, pp. 447 –462.

[7] D. Shepard, J. Bhatti, and T. Humphreys, "Drone hack: Spoofing attack demonstration on a civilian unmanned aerial vehicle," *GPS World*, August 2012.

[8] H. Teso, "Aicraft hacking," in *Fourth Annual HITB Security Conference in Europe*, 2013.

[9] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, 1973.

[10] Y. Zhang and R. West, "Process-aware interrupt scheduling and accounting," in *Proc. of the 27th IEEE International Real-Time Systems Symposium*, 2006.

[11] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2008, pp. 137–146.

[12] J. Hansen, S. A. Hissam, and G. A. Moreno, "Statistical-based wcet estimation and validation," in *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[13] A. Burns and S. Edgar, "Predicting computation time for advanced processor architectures," in *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on.* IEEE, 2000, pp. 89–96.

[14] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston et al., "Proartis: Probabilistically analyzable real-time systems," *ACM Transactions on Embedded Computing Systems*, 2013.

[15] K. Jiang, L. Batina, P. Eles, and Z. Peng, "Robustness analysis of real-time scheduling against differential power analysis attacks," in *2014 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2014, pp. 450–455.

[16] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The em side—channel (s)," in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2002, pp. 29–45.

[17] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[18] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems," in *Advances in Cryptology - CRYPTO 1996, 16th Annual International Cryptology Conference*, 1996.

[19] M. Völp, C.-J. Hamann, and H. Härtig, "Avoiding timing channels in fixed-priority schedulers," in *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2008, pp. 44–55.

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

[21] C.-Y. Chen, A. Ghassami, S. Mohan, N. Kiyavash, R. B. Bobba, and R. Pellizzoni, "Scheduleak: An algorithm for reconstructing task schedules in fixed-priority hard real-time systems," in *1st Workshop on Security and Dependability of Critical Embedded Real-Time Systems*, 2016.

[22] C.-Y. Chen, S. Mohan, R. Pellizzoni, and R. B. Bobba, "On scheduler side-channels in dynamic-priority real-time systems," *arXiv preprint arXiv:2001.06519*, 2019.

[23] C. Dwork, "Differential privacy: A survey of results," in *International conference on theory and applications of models of computation.* Springer, 2008, pp. 1–19.

[24] C. Dwork, A. Roth et al., "The algorithmic foundations of differential privacy," *Foundations and Trends® in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014.

[25] A. Easwaran, A. Chattopadhyay, and S. Bhasin, "A systematic security analysis of real-time cyber-physical systems," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC).* IEEE, 2017, pp. 206–213.

[26] C.-Y. Chen, M. Hasan, and S. Mohan, "Securing real-time internet-of-things," *Sensors*, vol. 18, no. 12, p. 4356, 2018.

[27] M. Yoon, S. Mohan, J. Choi, and L. Sha, "Memory heat map: Anomaly detection in real-time embedded systems using memory behavior," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[28] T. Xie and X. Qin, "Improving security for periodic tasks in embedded systems through scheduling," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 3, July 2007.

[29] M. Lin, L. Xu, L. T. Yang, X. Qin, N. Zheng, Z. Wu, and M. Qiu, "Static security optimization for real-time systems," *IEEE Transactions on Industrial Informatics*, vol. 5, no. 1, pp. 22–37, Feb 2009.

[30] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, June 2018, pp. 1–6.

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

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

[33] M. Hasan, S. Mohan, R. B. Bobba, and R. Pellizzoni, "Exploring opportunistic execution for integrating security into legacy hard real-time systems," in *2016 IEEE Real-Time Systems Symposium (RTSS).* IEEE, 2016, pp. 123–134.

[34] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, "A design-space exploration for allocating security tasks in multicore real-time systems," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 2018, pp. 225–230.

[35] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, "Contego: An adaptive framework for integrating security tasks in real-time systems," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, 2017.

[36] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo, "S3a: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems," in *Proceedings of the 2nd ACM international conference on High confidence networked systems.* ACM, 2013, pp. 65–74.

[37] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha, "Securecore: A multicore-based intrusion detection architecture for real-time embedded systems," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, 2013, pp. 21–32.

[38] F. Abdi, C.-Y. Chen, M. Hasan, S. Liu, S. Mohan, and M. Caccamo, "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, 2018, pp. 10–21.

[39] F. Abdi, C.-Y. Chen, M. Hasan, S. Liu, S. Mohan, and M. Caccamo, "Preserving physical safety under cyber attacks," *IEEE Internet of Things Journal*, 2018.

[40] N. Tsalis, E. Vasilellis, D. Mentzelioti, and T. Apostolopoulos, "A taxonomy of side channel attacks on critical infrastructures and relevant systems," in *Critical Infrastructure Security and Resilience.* Springer, 2019, pp. 283–313.

[41] S. Kadloor, N. Kiyavash, and P. Venkitasubramaniam, "Mitigating timing side channel in shared schedulers," *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1562–1573, June 2016.

[42] X. Gong and N. Kiyavash, "Quantifying the information leakage in timing side channels in deterministic work-conserving schedulers," *IEEE/ACM Trans. Netw.*, vol. 24, no. 3, pp. 1841–1852, June 2016.

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

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

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

[46] S. Kadloor, N. Kiyavash, and P. Venkitasubramaniam, "Mitigating timing based information leakage in shared schedulers," in *2012 Proceedings IEEE INFOCOM.* IEEE, 2012, pp. 1044–1052.

[47] X. Gong, N. Kiyavash, and P. Venkitasubramaniam, "Information theoretic analysis of side channel information leakage in fcfs schedulers," in *2011 IEEE International Symposium on Information Theory Proceedings.* IEEE, 2011, pp. 1255–1259.

[48] S. Liu, N. Guan, D. Ji, W. Liu, X. Liu, and W. Yi, "Leaking your engine speed by spectrum analysis of real-time scheduling sequences," *Journal of Systems Architecture*, 2019.

[49] M. Yoon, S. Mohan, C. Chen, and L. Sha, “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)*, April 2016, pp. 1–12.

[50] H. Baek and C. M. Kang, “Scheduling randomization protocol to improve schedule entropy for multiprocessor real-time systems,” *Symmetry*, vol. 12, no. 5, p. 753, 2020.

[51] K. Krüger, M. Völp, and G. Fohler, “Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems,” in *30th Euromicro Conference on Real-Time Systems (ECRTS)*, 2018, pp. 22:1–22:17.

[52] M. Nasri, T. Chantem, G. Bloom, and R. M. Gerdes, “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, 2019, pp. 103–116.

[53] K. Chatzikokolakis, M. E. Andrés, N. E. Bordenabe, and C. Palamidessi, “Broadening the scope of differential privacy using metrics,” in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2013, pp. 82–102.

[54] J. Cortés, G. E. Dullerud, S. Han, J. Le Ny, S. Mitra, and G. J. Pappas, “Differential privacy in control and network systems,” in *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE, 2016, pp. 4252–4272.

[55] Z. Huang, Y. Wang, S. Mitra, and G. E. Dullerud, “On the cost of differential privacy in distributed control systems,” in *Proceedings of the 3rd international conference on High confidence networked systems*, 2014, pp. 105–114.

[56] Y. Wang, Z. Huang, S. Mitra, and G. E. Dullerud, “Differential privacy in linear distributed control systems: Entropy minimizing mechanisms and performance tradeoffs,” *IEEE Transactions on Control of Network Systems*, vol. 4, no. 1, pp. 118–130, 2017.

[57] F. Liu, “Generalized gaussian mechanism for differential privacy,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 4, pp. 747–756, 2018.

[58] J. He and L. Cai, “Differential private noise adding mechanism and its application on consensus,” *arXiv preprint arXiv:1611.08936*, 2016.

[59] C. Tankard, “Advanced persistent threats and how to monitor and deter them,” *Network Security*, vol. 2011, no. 8, pp. 16–19, 2011.

[60] N. Virvilis and D. Gritzalis, “The big four - what we did wrong in advanced persistent threat detection?” in *2013 International Conference on Availability, Reliability and Security*, Sep. 2013, pp. 248–254.

[61] N. Falliere, L. Murchu, and E. C. (Symantec), “W32.stuxnet dossier,” http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf, 2011.

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

[63] J.-S. Pleban, R. Band, and R. Creutzburg, "Hacking and securing the ar. drone 2.0 quadcopter: investigations for improving the security of a toy," in *IS&T/SPIE Electronic Imaging.* International Society for Optics and Photonics, 2014, pp. 90 300L–90 300L.

[64] F. Samland, J. Fruth, M. Hildebrandt, T. Hoppe, and J. Dittmann, "Ar. drone: security threat analysis and exemplary attack to track persons," in *Proceedings of The International Society for Optical Engineering (SPIE)*, vol. 8301, 2012.

[65] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An EDF scheduling class for the Linux kernel," in *Real-Time Linux Wkshp*, 2009.

[66] jDrones, "ArduPilot Autopilot Suite," Jan. 2019. [Online]. Available: http://ardupilot.org/

[67] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' Track at the RSA Conference.* Springer, 2006, pp. 1–20.

[68] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel." *IACR Cryptology ePrint Archive*, vol. 2002, p. 169, 2002.

[69] "Erika Enterprise," http://erika.tuxfamily.org/drupal.

[70] "Real-time executive for multiprocessor systems (RTEMS)," https://www.rtems.org.

[71] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," in *European Symposium on Research in Computer Security*, 1998.

[72] S. K. Baruah, "Resource sharing in EDF-scheduled systems: A closer look," in *IEEE RTSS*, 2006, pp. 379–387.

[73] L. George, N. Rivierre, and M. Spuri, "Preemptive and non-preemptive real-time uniprocessor scheduling," INRIA, https://hal.inria.fr/inria-00073732/file/RR-2966.pdf, Tech. Rep., 1996, [Online].

[74] M. Spuri, "Analysis of deadline scheduled real-time systems," INRIA, https://hal.inria.fr/inria-00073920/file/RR-2772.pdf, Tech. Rep., 1996, [Online].

[75] H. S. Chwa, K. G. Shin, and J. Lee, "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, 2018, pp. 327–337.

[76] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms," *Real-Time Systems*, vol. 23, no. 1-2, pp. 85–126, 2002.

[77] P. Marti, C. Lin, S. A. Brandt, M. Velasco, and J. M. Fuertes, "Optimal state feedback based resource allocation for resource-constrained control tasks," in *25th IEEE International Real-Time Systems Symposium.* IEEE, 2004, pp. 161–172.

[78] F. D. McSherry, "Privacy integrated queries: an extensible platform for privacy-preserving data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, pp. 19–30.

[79] F. Liu, "Statistical properties of sanitized results from differentially private laplace mechanism with univariate bounding constraints," *arXiv preprint arXiv:1607.08554*, 2016.

[80] N. Holohan, S. Antonatos, S. Braghin, and P. Mac Aonghusa, "The bounded laplace mechanism in differential privacy," *arXiv preprint arXiv:1808.10410*, 2018.

[81] The Linux Foundation, "The Real Time Linux Collaborative Project," Jan. 2019. [Online]. Available: https://wiki.linuxfoundation.org/realtime/

[82] I. Esnaola-Gonzalez, M. Gómez-Omella, S. Ferreiro, I. Fernandez, I. Lázaro, and E. García, "An iot platform towards the enhancement of poultry production chains," *Sensors*, vol. 20, no. 6, p. 1549, 2020.

[83] B. M. R. Lima, L. C. S. Ramos, T. E. A. de Oliveira, V. P. da Fonseca, and E. M. Petriu, "Heart rate detection using a multimodal tactile sensor and a z-score based peak detection algorithm," *CMBES Proceedings*, vol. 42, 2019.

[84] N. Vreman, R. Pates, K. Krüger, G. Fohler, and M. Maggio, "Minimizing side-channel attack vulnerability via schedule randomization," in *2019 IEEE 58th Conference on Decision and Control (CDC).* IEEE, 2019, pp. 2928–2933.

[85] M. Bertogna and S. Baruah, "Limited preemption EDF scheduling of sporadic task systems," *IEEE Trans. on Ind. Info.*, vol. 6, no. 4, pp. 579–591, 2010.

[86] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *RTS Journal*, vol. 30, no. 1-2, pp. 129–154, 2005.