TIMING SIDE-CHANNELS IN REAL-TIME EMBEDDED SYSTEMS

by

VIJAY BANERJEE

A dissertation submitted to the Graduate Faculty of

University of Colorado Colorado Springs

in partial fulfillment of the

requirement for the degree of

Doctor of Philosophy

Computer Science

2026

This dissertation for Doctor of Philosophy degree by

Vijay Banerjee

has been approved for the

Department of Computer Science

by

Gedare Bloom , Chair

Yanyan Zhuang

Monowar Hasan

Terrance E. Boult

Xi Tan

April 29, 2025

Date

Banerjee, Vijay (Ph.D., Computer Science)

Timing Side-Channels in Real-Time Embedded Systems.

Dissertation directed by Associate Professor Gedare Bloom.

ABSTRACT

Real-time embedded systems are often employed in safety-critical fields such as avionics, automotive, and medical equipment, where stringent timing constraints are essential. Due to these constraints, tasks within the system are executed according to a predictable timeline. This dissertation examines the timing of side-channel attacks that exploit this predictability in real-time systems and proposes practical solutions for mitigating such threats. The research delves into side-channel attacks in real-time environments. The novel attack model introduced in the dissertation bypasses state-of-the-art side-channel defense in hierarchical real-time systems. The evaluation of this attack demonstrates its effectiveness, enabling precise targeting of victims with a precision exceeding 70% under normal system load.

The dissertation also focuses on improving the security posture of existing real-time operating systems. It presents a strategy for updating legacy software components to bolster the security posture of critical infrastructure against side-channel attacks that target known vulnerabilities within the code. This strategy employs a modularization-based approach, allowing legacy systems to coexist with upgraded software components during the transition period, facilitating the integration of security updates for legacy components without necessitating significant system downtime. Additionally, the dissertation outlines recovery techniques designed to ensure the timely restoration of the system to a functional state

following a successful attack. The theoretical analysis establishes timing bounds on the recovery process and defines the feasibility conditions. The results of this dissertation have led to the development of software packages utilized by prominent institutions, including specific Department of Energy national laboratories. The artifacts generated from this research have been open-sourced and made available to the public for free use.

ACKNOWLEDGMENTS

I want to express my heartfelt gratitude to Professor Gedare Bloom for inspiring me to pursue a Ph.D. Your invaluable advice, guidance, and mentorship throughout this journey have been instrumental. Thank you for encouraging me to explore my ideas and for helping me transform them into tangible research outcomes.

I extend my thanks to Dr. Sena Hounsinou for being an exceptional mentor, co-author, and friend. You consistently provided answers to questions that couldn't be easily found online and offered critical guidance on everything from navigating graduate school to managing the challenges of being an international student in the USA.

I am grateful to Dr. Monowar Hasan for your continuous input on my research ideas and for enhancing my research paper submissions through your insightful reviews and technical feedback. I have learned a lot from our technical discussions and the brainstorming sessions.

I sincerely thank Dr. Yanyan Zhuang for underscoring the importance of consistency in research, which has greatly improved both this dissertation and my other academic publications.

I would also like to thank Dr. Terry Boult and Dr. Xi Tan for serving on my dissertation committee and for sharing insights on improving the evaluation of this dissertation's contributions.

I want to thank all my co-authors who contributed to the research and development that resulted in this dissertation. Your contributions strengthened the papers and respective scientific contributions made by each publication.

I am genuinely grateful to be part of the wonderful RTEMS community. This incredible group has significantly shaped my understanding of real-time systems, and I feel fortunate to learn from such talented individuals. A special thanks to Dr. Joel Sherrill and Chris Johns for their invaluable guidance during my research and their unwavering support in helping me push my work upstream. Your insights have made a profound difference, and I sincerely appreciate it.

Pursuing a Ph.D. is a significant challenge that tests one's mental, intellectual, and emotional limits. I could not have navigated this process without the immense support of all the healthcare professionals who helped me stay alive, sane, and functional throughout the process. Starting from COVID-19 to an acute mental crisis, I was able to survive through these times due to their tireless support. I want to thank Dr. Brandon Baker for ensuring that my mental barriers do not stop me from contributing to science. I would also like to thank Alex David for teaching me the essential elements of a healthy lifestyle and for helping me actively stay on track with those elements.

My achievements directly result from the immense support I have received from my family. I will forever be grateful and indebted to my mom, Snigdha Banerjee, for single-handedly fighting against all odds to support me in all endeavors, including academics, sports, and music. Thank you for never giving up on me.

Thanks to my brother Somnath Banerjee for being an inspiration and role model throughout my childhood. I have always looked up to you.

I also want to thank my dad, Krishnanath Banerjee. I wish you were here to read this dedication. I hope I made you proud. Rest in peace.

Finally, I want to thank my wife, Ratnabali. Your creativity inspires me, your wit humbles me, your resilience awes me, and your belief in me gives me the courage to embark on journeys like the one that resulted in this dissertation. Thank you.

For my wife Ratnabali. I love you.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

TABLE

LIST OF FIGURES

FIGURE

CHAPTER I

INTRODUCTION

Real-time embedded systems (RTES) represent a specialized subclass of computing systems characterized by stringent constraints on timing parameters. Frequently deployed in safety-critical domains, such as automotive applications, aerospace software, and medical devices, these systems require a timely execution of tasks to avert potentially catastrophic outcomes. The proliferation of RTES within essential infrastructures—including transportation, energy, and defense—exacerbates their vulnerability to cyber threats. The advent of the Internet of Things (IoT) further compounds this risk, as RTES interface with diverse internal and external networks, heightening susceptibility to remote attacks. Prior research has indicated that conventional security mechanisms designed for general-purpose systems often prove unsuitable for RTES, primarily due to their rigorous timing requirements and resource constraints. Consequently, any proposed security enhancements necessitate comprehensive timing and overhead analyses before integration into existing frameworks [2].

Historically, RTES designs have prioritized functionality over security considerations, resulting in numerous legacy systems with ambiguous security postures. The landscape of potential vulnerabilities, attack vectors, and defensive strategies related to RTES remains inadequately understood. In critical environments, the challenges of upgrading these legacy systems are significant, as transitioning to modern implementations demands extensive engineering efforts and may inadvertently introduce new vulnerabilities alongside potential reliability issues. This dissertation investigates the security challenges intrinsic to RTES, focusing on timing side-channel attacks that exploit known vulnerabilities and threaten these systems' confidentiality, integrity, and availability—collectively referred to as the *CIA triad*.

The vulnerability associated with timing predictability in RTES has been extensively documented in the literature. While existing studies advocate for randomization-based techniques to mitigate the likelihood and impact of such attacks, these methodologies have proven ineffective due to RTES's intrinsic timing and resource constraints. This research critically evaluates the limitations of current defenses and introduces a novel timing-based side-channel attack capable of bypassing these defense techniques.

The dissertation also establishes viable system recovery techniques and engineering solutions that can enhance and facilitate the recovery of systems under attack. The presented work has been evaluated from two perspectives: the repercussions of the attacks and the system overhead associated with the proposed defenses. The artifacts developed throughout this dissertation are available under open source license, promoting unrestricted access.

1.1 Research Questions

This dissertation focuses on the following three research questions and addresses these questions respectively in Chapters III, IV, and V.

RQ1 What methodologies can attackers employ to evade defensive measures against timing side-channels in real-time systems?

RQ2 What techniques can facilitate upgrading legacy software components with minimal system downtime?

RQ3 What strategies can be employed to restore the system to a trusted state within an acceptable time frame?

1.2 Research Contributions

This dissertation systematically addresses the research questions outlined. Chapter III explores RQ1 through a novel attack approach that alters the execution parameters of adversary tasks to enhance the side-channel inference capabilities of target victim tasks. The attack, referred to as NOSYNEIGHBOR, proves effective against the prevailing schedule randomization techniques. An evaluation of NOSYNEIGHBOR is conducted using a discrete task simulator alongside a case study highlighting its efficacy against the state-of-the-art defense technique BLINDER [3].

Upgrading outdated software components and adhering to updated security protocols is paramount for mitigating the impact of attacks and strengthening the security posture of systems. This necessity motivates RQ2, which is addressed in Chapter IV through a

modularization-based strategy for upgrading components within a monolithic kernel widely utilized in critical infrastructure sectors. The presented technique enables the continued operation of legacy software components while facilitating the testing and evaluating new components without causing significant system downtime. This chapter systematically migrated over $270,000$ lines of code, which were made freely available as open source libraries [4, 5]. These libraries were contributed to the RTEMS project that multiple esteemed institutions use in safety-critical domains. Numerous national laboratories have actively leveraged the research's outcomes to transition to newer software components within their existing infrastructures.

Recognizing that no system can achieve complete security, it is crucial to implement recovery mechanisms that restore systems to a functional state post-attack. Chapter V presents a secure boot-based recovery technique that allows the Real-Time Embedded Systems (RTES) to be periodically rebooted into a trusted state. This chapter addresses RQ3 by presenting a classical response time analysis incorporating the overheads due to periodic recovery and secure reboot. The theoretical analysis applies to any periodic recovery methodology in a real-time system. The response time analysis shows minimal impact on the task schedulability, proving the feasibility of the secure boot in any real-time system with a periodic recovery mechanism. The theoretical insights offered here can also inform future research evaluating the feasibility of alternative periodic recovery techniques.

CHAPTER II

RELATED WORK

2.1 Side-Channel Attacks in Real-Time Embedded Systems

Side-channel attacks are confidentiality attacks that exploit a system's behavior to leak sensitive information. These attacks are frequently combined with other attacks that utilize the confidential information obtained through side channels. In RTES, attackers can exploit timing information from an executing process to carry out additional attacks, such as denial of service (DoS). This section discusses various existing side-channel attacks and their impact on RTES.

2.1.1 Parameter Inference.

In ScheduLeak [6, 7], the attacker leverages an unmanned aerial vehicle's idle task to collect observations about the system's schedule from which the busy intervals and the victim task's initial offset and future arrival times are inferred. Hounsinou et al. [8] recorded controller area network (CAN) traffic to reconstruct the CAN schedule using the CAN response time analysis. They identified patterns of CAN messages that are expected to

precede the victim to queue the attack message with heuristics to improve the success of their attack. Others have shown how circular auto-correlation, fast Fourier Transform [9], and periodogram [10] can infer task parameters from execution traces. These approaches differ in accuracy, particularly when faced with system uncertainties like release jitter, and have not been implemented in hierarchical systems. Furthermore, the accuracy of these techniques is static, meaning they conduct a side-channel inference only once at the end, which makes them a transient attack. In contrast, the NOSYNEIGHBOR attack introduced by this dissertation adjusts attack parameters and enhances inference accuracy over time. Therefore, NOSYNEIGHBOR can be classified as a persistent attack.

2.1.2 Randomization-based Defense Techniques.

Schedule randomization [11–14] aims to reduce the apparent determinism of real-time systems to prevent an adversary from deducing timing parameters. However, Nasri et al. [15] argued that these approaches might not provide enough protection while stating that isolation techniques (including virtualization) might directly solve the security-relevant problems of schedule information leakage. A similar line of work, Blinder [3] and TimeDice [16], use randomization-based solutions to mitigate the orchestration of multiple tasks. However, through a proof-of-concept implementation, we demonstrate that the NOSYNEIGHBOR can successfully infer the victim's timing even in the presence of these defense techniques. NOSYNEIGHBOR's design makes it resilient to simple randomization due to the real-time workloads' contained deadline that limits the randomization-based defense's impact.

2.2 Real-Time Recovery[1]

Trusted computing aims to protect systems against integrity attacks by providing an outlet for a root of trust, uniquely identifying a platform. The Simplex architecture has been used for fault tolerance in control systems utilizing untrusted logic and an isolated safety controller. Approaches based on System-level Simplex architecture [17] and restart-based (both revival [18, 19] and rejuvenation [20, 21]) approaches run the safety controller and decision module on dedicated hardware. These methods add a safety guarantee to the Simplex-based architecture.

Using System-level Simplex architecture, Abdi et al. [22] proposed a restart-based recovery approach for the complex subsystem when software faults are detected. Further, Abdi et al. [20,21,23] proposed a framework to periodically restart the platform to improve the safety of real-time systems and provide a system-wide restart-based approach that provides a formal guarantee of system safety. However, these works do not provide proof of timing guarantee and feasibility analysis for real-time systems. In contrast, we provide a framework for the feasibility analysis of restart-based recovery and integrate secure boot into the restart to guarantee a trusted system state after every restart.

Romagnoli et al. [24] proposed a recovery technique based on software refresh that guarantees the controller integrity and safety. However, recovery does not prevent attacks from occurring again. We address this shortcoming by adding a secure boot as a part of every restart. We use the simplex-based architecture [25–28] using decision procedures that

[1] Portions of this section were published in [1]

provide fault-tolerant and low-overhead solutions to choose control commands between the complex controller and safety controller to improve system reliability.

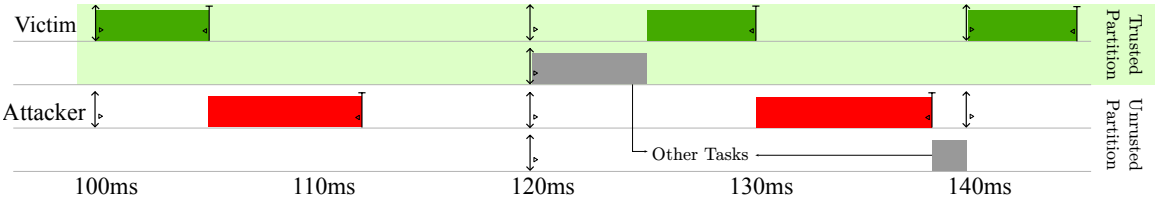
Diversification-based security leverages the system's physical properties to introduce execution path randomness after every restart [29, 30]. Configuration files, memory location, and hardware state are diversified to decrease the exposure of system vulnerabilities after periodic reset operations, which prevents persistent attacks. However, the overhead of such an approach can become infeasible for RTES. This dissertation uses deterministic system execution paths to enable security without compromising the predictability of the system through theoretical bounds on the security mechanisms.

CHAPTER III

NOVEL SIDE-CHANNEL ATTACK IN HIERARCHICAL REAL-TIME SYSTEMS[1]

This chapter presents NOSYNEIGHBOR, a novel attack against hierarchical systems that can find execution windows of a victim task in an isolated partition. In a real-time system, tasks are schedulable entities with predictable execution times and guaranteed deadlines. NOSYNEIGHBOR exploits this timing constraint of real-time systems to infer the execution of tasks across the boundaries of isolated partitions in hierarchical systems. An attacker can use NOSYNEIGHBOR to effectively evade the state-of-the-art defense techniques for hierarchical systems that depend on schedule randomization [3,16]. These defense techniques assume tasks have static parameters (execution time and period), i.e., they presume malicious tasks exhibit the same execution behavior over time. In practice, however, the execution behavior of tasks in real systems can vary widely over time. NOSYNEIGHBOR uses malicious *adaptive* tasks to draw timing inferences of the victims that improve iteratively throughout the attack execution. This adaptive inference fundamentally differs from

[1]THIS CHAPTER IS UNDER REVIEW AT ACM TRANSACTIONS ON CYBER-PHYSICAL SYSTEMS



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

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

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

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

Figure 3.1: A demonstration of adaptive task on Litmus-RT.

the state-of-the-art attack models that use a single-stage timing inference, which is the final inference.

Hierarchical systems, including virtualization-based systems, use shared hardware resources for multiple tasks. Hardware sharing makes them vulnerable to side-channel information flow between the tasks. Embedded systems are especially prone to such vulnerabilities due to limited hardware resources. In these systems, obtaining precise timing information of security-critical tasks can lead to more serious attacks such as cache side-channel attacks [31]. By using the adaptive execution model, NOSYNEIGHBOR can draw more precise timing information of a victim task than previously known, which increases the effectiveness of existing time-sensitive attacks.

Schedule-based attacks have been extensively studied in real-time systems [6–8, 32]. These attacks can be classified as anterior, posterior, concurrent, or pincer, depending on the temporal proximity of the attacker's execution in relation to the victim [15]. To determine

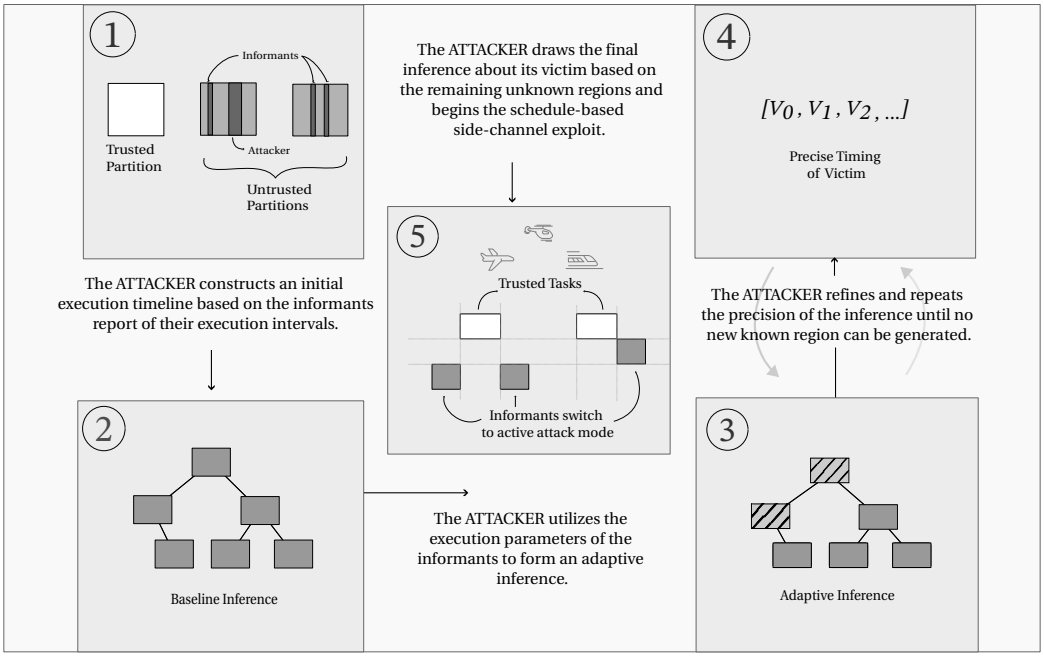


Figure 3.2: The NOSYNEIGHBOR Attack: On each phase of the attack, the malicious tasks *adapt* their parameters and improve attack precision. NOSYNEIGHBOR utilizes the precise placement of malicious tasks to switch into an active attack. Section 3.2 discusses these steps in detail.

the best timing to execute the attacker's tasks, the attacker typically starts by observing the system. During this observation, the attacker collects measurements about the system, enabling them to infer the system's parameters in general and the timing parameters of the victim task in particular. However, the existing literature does not consider *adaptive execution* of the attacker to improve the attack inference over time. NOSYNEIGHBOR uses adaptive execution to evade state-of-the-art defense techniques based on schedule randomization.

Figure 3.1 illustrates the basic concepts and feasibility of adaptive attack execution. The execution time of the attacker task in Figure 3.1a is different in two subsequent executions. In the second iteration of the task that starts at 130 ms, the execution time is increased by 1 ms. This extended 1 ms execution time reflects that no higher-priority tasks were ex-

ecuted during that time. Such extended execution is an example of adapted execution used by NOSYNEIGHBOR to infer execution information of higher priority tasks through side-channel. Section 3.2.4.1 describes the *extended execution* used in Figure 3.1a. Figure 3.1b shows the code modification used by the attacker task of Figure 3.1a. Such modification can be maliciously inserted into the program by an adversary to enable the adaptive execution of a task based on external inputs. An adversary can insert such malicious programs into the system by using existing code injection or supply chain attacks [33, 34]. NOSYNEIGHBOR uses adaptive tasks similar to Figure 3.1b, in which we show that these malicious tasks are not required to be inside a trusted partition, and an adversary can use these tasks from an untrusted and unauthorized partition to derive the precise execution times of the victim. The attack precision depends on the relation between the actual execution window of a task and the execution window inferred by NOSYNEIGHBOR.

To evaluate the success of the NOSYNEIGHBOR attack, I used the concept of attack effective window (AEW), introduced by Chen et al., for side-channel attack evaluation [35, 36]. An AEW refers to a time frame specific to each victim's execution window. For any given victim execution window, the relative AEW will encompass a superset that includes the interval during which the victim's execution occurs. In the literature, an attack is considered effective only if executed within the victim's AEW. Although the term AEW implies that an attack will always be successful within this window, the resultant intervals do not guarantee an impact on the system in the presence of an active attack. Hence, a more precise term would be *attack potential window*. Nevertheless, I will continue to use the term AEW throughout this dissertation for the sake of consistency with existing literature. Further description of the AEW and its use in the evaluation is provided in Section 3.3

An overview of the NOSYNEIGHBOR attack is shown in Figure 3.2. First, the NOSY-NEIGHBOR attacker constructs an initial timeline of the malicious tasks ① and builds a baseline inference ② of the victim task. Next, it adapts the execution parameters of the malicious tasks ③ to increase its knowledge of the system's schedule and improve the inference of the victim's execution. Over time, its inference tightens more precisely ④ on the victim's actual execution timing. Eventually, the NOSYNEIGHBOR terminates its inference ⑤ when it decides it has enough information about the victim task to conduct a schedule-based attack.

3.1 Models and Assumptions

This section details the system and threat model assumed to demonstrate and evaluate NOSYNEIGHBOR. I also present assumptions related to the attacker's capabilities and the vulnerabilities that an attacker can exploit.

3.1.1 System Model

I assume a uniprocessor real-time hierarchical system. The hierarchy is provided by virtualization technology, such as a hypervisor that creates K partitions in the system. Among these partitions, at least one is trusted, and at least two are assigned to untrusted tasks. The attacker utilizes inter-partition collusion of tasks from distinct untrusted partitions to generate precise timing inference of a targeted trusted task. I denote the targeted task as the *victim task*. The system model assumption is aligned with the state-of-the-art techniques addressing side-channels [3]. Although NOSYNEIGHBOR can be executed from

a single untrusted partition if the system has only two partitions, the attack is most effective when the malicious tasks are spread across multiple partitions. We also assume that the hypervisor is trustworthy and secure from attacks, and thus, it correctly schedules partitions. A *global scheduler*, a part of the hypervisor, is responsible for scheduling the partitions based on their budget. Note that the global scheduler has no information about the individual tasks inside the partitions. Each partition has a *local scheduler* to handle the tasks inside that partition. As demonstrated in Figure 3.1a, the *P-RES* scheduler in LITMUS-RT [37] is an example of a hierarchical scheduler that matches the system model here.

Task Model. Each partition is denoted by Π^k, and has a priority of $k \in \{0, \ldots, K-1\}$. Additionally, Π^k is characterized by a period of T^k and an execution capacity of C^k within each execution period. We use the term *major frame*, denoted M, as commonly defined in hierarchical systems as the least common multiple of all T^k. A task of priority i in partition Π^k is denoted by τ_i^k. Each task τ_i^k has a minimum inter-arrival time of t_i^k, after which a new iteration or job of τ_i^k is released. A job $\tau_{i,j}^k$ is the j'th iteration of task τ_i^k. Each task has a worst-case execution time (WCET) of c_i^k and a relative deadline of d_i^k.

3.1.2 Threat Model

The NOSYNEIGHBOR adversary aims to launch a schedule-based side-channel attack on a *victim task* executing in a trusted partition. We assume that the trusted partition is scheduled at a higher priority than the untrusted partitions, as is typical in hierarchical systems comprising software components from multiple vendors [38]. Although the security-criticality and priority are not generally coupled in general-purpose systems, the time-critical tasks such as navigation are generally placed at a higher priority partition

in automotive-grade OS such as QNX [39]. Knowing the execution times of these time-critical tasks can enable an adversary to execute time-sensitive attacks like denial of service.

We assume that the adversary can infiltrate and control *some* insecure application tasks executing in untrusted partitions of the hierarchical system remotely. Specifically, we assume that once the attacker has a foothold on a task, they are capable of controlling the task's execution parameters and configuring inter-partition communication for that task. This system assumption is common in real-time systems, and one can leverage existing attacks to achieve this capability [40]. We refer to the set of such tasks under the attacker's control as *malicious tasks* and denote them by $\mathcal{T}_A$. One of these tasks is designated as the NOSYNEIGHBOR, and the remainder tasks are designated *informants*—they are formally defined in Definitions III.1 and III.2 in Section 3.2.

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

3.2 The NOSYNEIGHBOR Attack

This section details each step of the NOSYNEIGHBOR attack. The NOSYNEIGHBOR attack strategy uses two categories of tasks, NOSYNEIGHBOR task and *informant* tasks, which are defined below. These tasks are executed in the untrusted partition and have a lower priority than the victim task.control these observer and informant tasks executing in untrusted partitions and collect their execution intervals to infer the timing information of the (higher priority)

3.2.1 Definitions

Definition III.1 (The NOSYNEIGHBOR task)**.** *In the* NOSYNEIGHBOR *attack, a primary malicious task is responsible for recording, analyzing, and adapting the execution parameters of all the malicious tasks in the system. The* NOSYNEIGHBOR *task holds the following properties to execute a successful attack:*

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

Definition III.2 (The Informants)**.** *The* informants *are a set of malicious tasks that are coordinated by the* NOSYNEIGHBOR *task. Each informant holds the following properties:*

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

I use the notation $I^k_{i,j,n}$ to denote an informant task executed during the n'th major frame in partition Π_k, with task id i, and job id j.

Definition III.3 (Known and Unknown Regions)**.** *A* known region *is any time interval in a major frame during which at least one informant has executed. In contrast, an* unknown region *is an interval in the major frame during which no informant has executed since the beginning of the attack (either because no informant has been scheduled during that*

interval or an execution by an informant was unsuccessful due to preemption). At the end of the n'th major frame, we denote the total known region as ζ_n, and unknown regions are denoted ζ'_n. ζ'_n is considered the attack inference after n major frames.

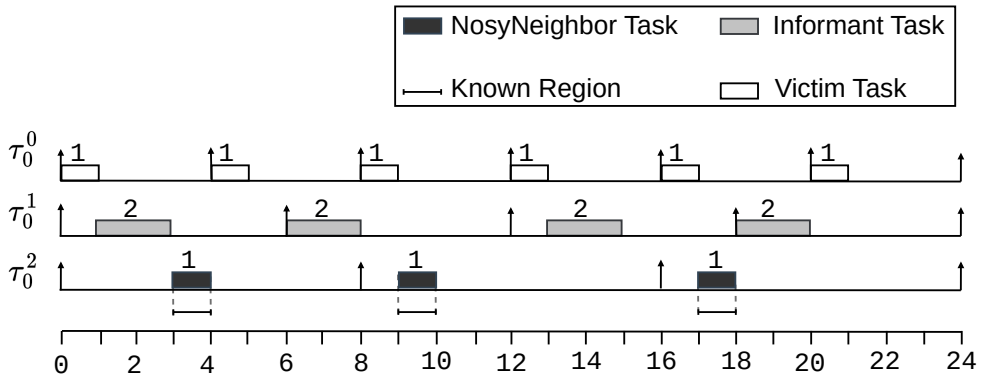
3.2.2 Overview

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

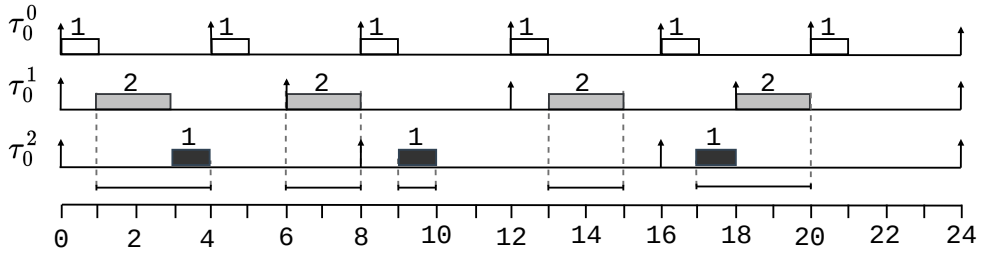
1. The NOSYNEIGHBOR task generates an initial set of parameters for each informant task.
2. The informants execute based on the initial parameters and report the execution intervals of all their jobs throughout the major frame. The NOSYNEIGHBOR task constructs a baseline inference with these reports.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

The steps 1 and 2 form a *baseline inference* (Section 3.2.3). NOSYNEIGHBOR uses this baseline inference to decide *parameter adaptation* for the informant tasks (Section 3.2.4) in step 3. Upon completing the previous steps, NOSYNEIGHBOR can use the timing inference to conduct schedule-based attacks (i.e., schedule exploitation). We illustrate these stages using the following example.

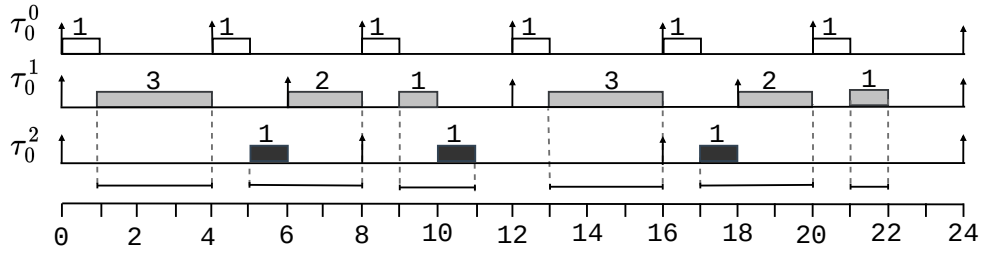
Example III.1. *Consider a three-partition hierarchical system with a major frame of $M = 24$ units as depicted in Figure 3.3. Task τ_0^2 is the* NOSYNEIGHBOR *task and τ_0^0 is the*



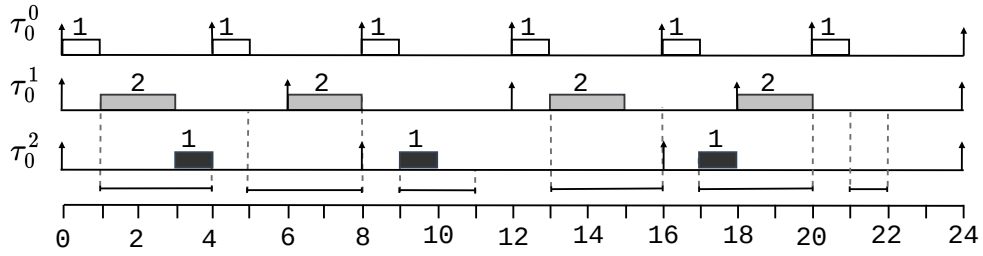
(a) Baseline inference of the NOSYNEIGHBOR task τ_0^2 with no collusion.



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



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



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

Figure 3.3: The NOSYNEIGHBOR Adaptive Execution: the attacker adapts the known regions by modifying the parameters of the informant task τ_0^1, which improves the inference of the execution pattern of the victim task τ_0^0.

victim. The baseline inference stage begins in Figure 3.3a. Here, τ_0^2*'s knowledge about the schedule is limited to the duration of its own execution (*$[3, 4)$*,* $[9, 10)$*, and* $[17, 18)$*). This leads to a poor inference of the execution pattern of victim* τ_0^0*. When* τ_0^1 *is used as an informant with an initial execution budget of* 2 *units (Figure 3.3b), the known regions include additional intervals* $[1, 3)$*,* $[6, 8)$*,* $[13, 15)$*, and* $[18, 20)$ *due to the successful (non-preempted) executions of* τ_0^1*.*

The parameter adaptation starts in the next major frame (Figure 3.3c). The NOSYNEIGHBOR*task extends the execution time of* τ_0^1 *to* 3 *units and subsequently reduces it to* 2 *units (Figure 3.3d) because the extension resulted in preemption of* $\tau_{0,1}^1$ *and* $\tau_{0,3}^1$ *(in Figure 3.3c). Altogether, this yields another expansion of the known regions by generating intervals* $[5, 6)$*,* $[10, 11)$*,* $[15, 16)$*, and* $[21, 22)$ *in Figure 3.3c. With this, the* NOSY-NEIGHBOR *task can compute the total known regions of* M *and deduce the unknown regions* $[0, 1)$*,* $[4, 5)$*,* $[6, 9)$*,* $[11, 13)$*,* $[16, 17)$*,* $[20, 21)$*, and* $[22, 24)$*. Six out of those seven inferred regions contain an execution of the victim task* τ_0^0*.*

3.2.3 Baseline Inference

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

to manage the recorded timestamps of the informant tasks and calculate the parameter adaptations for the next iterations of the attack.

3.2.3.1 Informant Data Representation

The NOSYNEIGHBOR attack requires storing and orchestrating the informant tasks' individual jobs over multiple major frames and analyzing their timing logs. This poses a challenge because of the number of jobs of each task that may participate in the attack. NOSYNEIGHBOR uses an interval tree [41,42] to handle the large number of execution logs collected by informant jobs. An interval tree is an ordered self-balancing tree where each node represents an interval. It is also an efficient structure for finding overlapping intervals, which is an important property for selecting the group of informants for adaptive execution. For m informants, the tree can be constructed and searched with a time complexity of $\mathcal{O}(m \log m)$ and $\mathcal{O}(\log m)$, respectively.

Definition III.4 (Informant Node and Informant Tree)**.** *An* informant node *represents a time interval that denotes the start and end times corresponding to a job's response time. We denote the informant node of an informant's job $\tau_{i,j}^k$ during nth major frame by $I_{i,j,n}^k$. The* informant tree I *is the interval tree constructed from all informant nodes. The informant tree is updated at the end of the nth major frame with new values of $I_{i,j,n}^k$.*

3.2.3.2 Handling Uncertainty and Schedule Randomization

Schedule-based randomization adds uncertainty to the inferred timeline from the attack. Such schedule uncertainties can also result from system noises and release jitters.

Depending on the randomization protocol used in the system, each job can have varying magnitudes of uncertainties.

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

In two subsequent major frames, the reported known regions overlap only if the shift in the execution is less than the execution cost of the informant node. Hence, for major frames n and $n-1$, the reported known regions overlap if the difference between release times of $\tau^k_{i,j,n}$ and $\tau^k_{i,j,n-1}$ is less than c^k_i. The newly reported region may yield a new known region, which is wider than the previously recorded known region. To expand the known regions from one major frame to the next, the union of the overlapping intervals is recorded in an interval tree. Thus, the informant node of overlapping intervals at the end of the nth major frame ($n \geq 1$) is defined as:

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

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

Similarly, the NOSYNEIGHBOR can evade randomization-based defense techniques by relying on multiple major frames to collect a baseline value for each $I^k_{i,j,n}$. The ran-

domness achieved by most randomization-based mitigation techniques is limited because of the time-bounds constraints of the real-time tasks. As a result, it is possible to reveal the predictable randomization patterns from the observation of multiple major frames.

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

1. $I_{i,j,n}^k \succ I_{i,j,n-1}^k$
2. $I_{i,j,n}^k \prec I_{i,j,n-1}^k$
3. $\{I_{i,j,n}^k\} \cap \{I_{i,j,n-1}^k\} \neq \emptyset$

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

3.2.3.3 Constructing the Baseline Inference

We take a heuristic approach to update the known regions after each major frame using Equation (III.1) by introducing additional informant nodes for cases 1) and 2) above

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

Algorithm 1 Forming the known and unknown set of regions

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

for the same job. Since NOSYNEIGHBOR aims to segment the timeline into known and unknown regions, adding more informant nodes will cover more regions of possible executions of informants. Algorithm 1 shows the steps to construct and update the informant tree over multiple major frames.

To construct the interval tree, Algorithm 1 compares the current interval ($I^k_{i,j,n}$) to the total execution time of the informant task, which indicates preemption (Line 4). Subsequently, the algorithm checks whether the previously recorded interval in the set of known regions ζ overlaps with the current interval ($I^k_{i,j,n}$) (Line 7). We check overlapping intervals to handle the variances and randomizations by segmenting the informant intervals to augment the known and unknown regions or expand the overlapping regions. On Line 9,

ζ is updated as the union of both intervals and inserted in the tree. Otherwise, $I^k_{i,j,n}$ is recorded in ζ' tree. The attacker then investigates whether the overlapping preempted intervals are due to the execution of a higher-priority secure task or due to the execution of a higher-priority informant task.

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

Algorithm 2 Deriving Inference from Unknown Regions

Input: I, ζ, ζ'
Output: V
$V = \emptyset$
for $\delta \in I$ **do**
    $executed \leftarrow 0$
    $O \leftarrow$ Overlap(I, δ)
    $\delta' \leftarrow$ NextNode(I, δ)[3]
    $V \leftarrow V \cup \{[\delta_{end}, \delta'_{start})\}$
    if $O \neq \emptyset$ & $\delta_{end} - \delta_{start} > \sum o \in O + \delta_{etime}$ **then**
        $budget \leftarrow \delta_{etime}$
        if $|O| = 1$ **then**
            $executed = \delta_{end} - \delta'_{end}$
            $remaining \leftarrow budget - executed$
        else
            for $i \in [1, O_{end})$ **do**
                $executed \leftarrow executed + (O_{i+1}.start - O_i.end)$
            $executed \leftarrow executed + \delta_{end} - O_{last}.end$
            $remaining \leftarrow budget - executed$
        if $O_1.start - \delta_{start} \neq remaining$ **then**
            $V \leftarrow V \cup \{[\delta_{start} + remaining, O_1.start\}$
return V

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

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

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

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

Equation (III.2) can yield interval endpoint values that are 0 for scenarios when the next executing task in the timeline is one of the informant tasks because the start and end times are consecutive. In these scenarios, δ_v is not added to the inference tree V. Line 8 in Algorithm 2 corresponds to Equation (III.2) above.

For ***preempted informants***, there can be two cases that are handled in Line 10: (1) multiple informant tasks overlap, and (2) no other informant task overlaps. In case (1), the

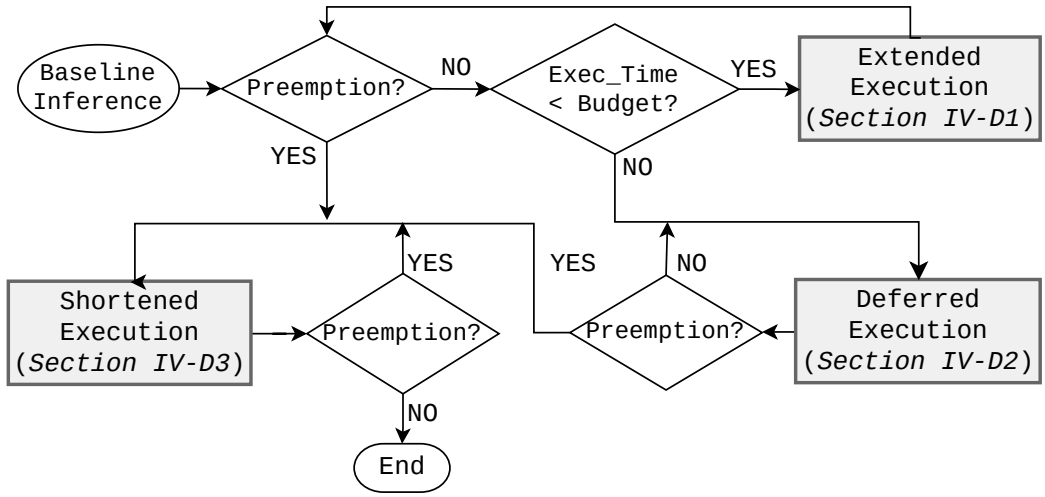


Figure 3.4: Informant Task Execution Model.

preemption is caused by one of the other informants, a higher-priority task, or both. To find the victim node inference in this case, we use the following equation for an arbitrary informant node $I(i) \in I$:

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

Equation (III.3) takes the interval of $I(i)$ and excludes (via $\setminus$) the intervals covered by other informant nodes. Thus, Δ is a set with intervals as elements. Each of these intervals is added to V' as an inference node. Line 16 uses Equation (III.3) to select the time regions where the informant task got preempted, leaving *gaps* in the execution phase that can be potential execution times of the victim. These gaps are then added to the inference tree V.

3.2.4 Parameter Adaptation

After forming the baseline inference, NOSYNEIGHBOR uses multiple adaptive strategies to adapt the execution parameters of informant tasks to improve the total timeline area

covered by all the intervals in V. The goal of these strategies is to *probe* the victim nodes by executing informant tasks during time intervals that should overlap with the victim nodes by extending, deferring, or shortening the execution of an informant, as shown in Figure 3.4. To effectively probe the victim nodes, NOSYNEIGHBOR first selects a subset of informant nodes and sends the modified task parameters to them for adaptive execution in the next major frame.

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

During the construction of the baseline inference, NOSYNEIGHBOR uses combinations of informant nodes to find gaps in execution times. To overlap adaptive execution with a victim node, $\delta_v \in V$, NOSYNEIGHBOR selects the nodes from I that were used for calculating the time window δ_v. NOSYNEIGHBOR uses these selected nodes to reduce the unknown regions using different strategies based on whether non-preempted or preempted informants generated the victim node. Each victim node is marked as *visited* when the maximum possible informant tasks have been attempted to execute within that interval. NOSYNEIGHBOR uses three strategies to *visit* the victim nodes: extended execution, deferred execution, and shortened execution. Adaptive execution finishes after the victim node intervals is not shortened by subsequent attack iterations, i.e., after one major frame

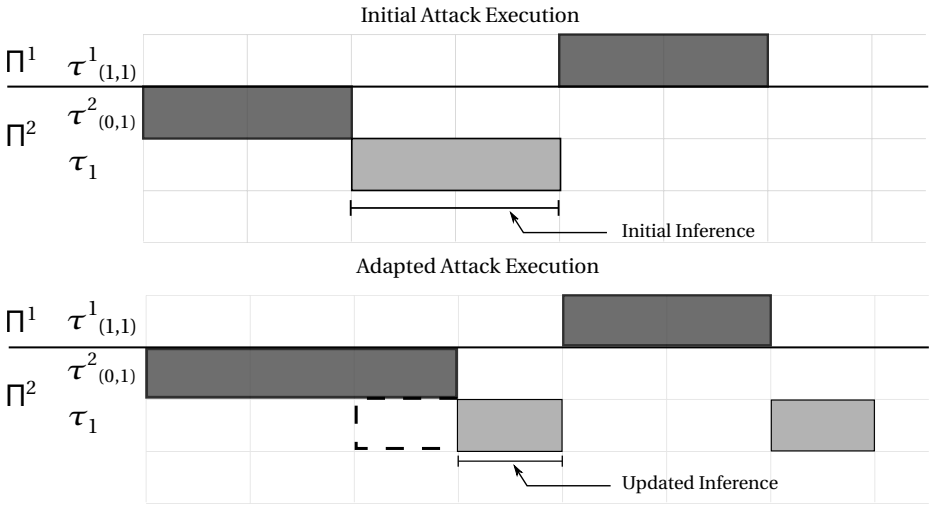


Figure 3.5: Extended execution of informant $\tau^1_{0,1}$.

with no change in the victim node. To keep track of the interval updates in each iteration of the attack, the structure of δ_v contains the list of informant tasks, and indicate if this node has been visited, and the victim interval (start to end time). The subsequent sections describe the three types of adaptation that NOSYNEIGHBOR uses.

3.2.4.1 Extended Execution

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

Extended execution of a task from a higher priority partition will block the execution of a task from a lower priority partition. Hence, if a victim node emerges due to a task from a partition of lower priority, extended execution of a higher priority task will block it, and

the victim node will be shortened or removed from the victim tree, improving precision. Example III.2 illustrates the extended execution strategy.

Example III.2. *Consider Figure 3.5 for this example. We denote the victim node as* τ_0^0. *Suppose* τ_0^0 *delayed the release of* $\tau_{0,1}^1$. *After the baseline inference,* NOSYNEIGHBOR *modifies the parameter of informant* $\tau_{0,1}^1$ *to extend its execution time. Due to the higher priority of* Π_2 *over* Π_1, *the informant node gets shifted due to the extended execution of* $\tau_{0,1}^1$. *Since the informant was able to execute for the entire window without preemption, we remove its informant node from the victim tree* V. *The resulting inference is smaller (more precise), and therefore, the likelihood of predicting the victim task's execution is greater.*

3.2.4.2 Deferred Execution

Each task in the system has a maximum execution budget that gets replenished at regular intervals. The tasks can be executed anywhere in the timeline as long as it does not exhaust the allocated budget. If the allocated budget is exhausted, the task has to wait for the next replenishment period to start execution again. NOSYNEIGHBOR uses this property of budgets to defer the execution of informants.

Deferred execution can be achieved by calling `sleep()` from the informant task to allow the informant to probe a victim node interval. This is highly effective if the informant has already exhausted its allocated budget. An example of deferred execution is shown in Figure 3.6. Similar to Example III.2, the victim node will be removed from V after the adapted execution of $\tau_{0,1}^1$.

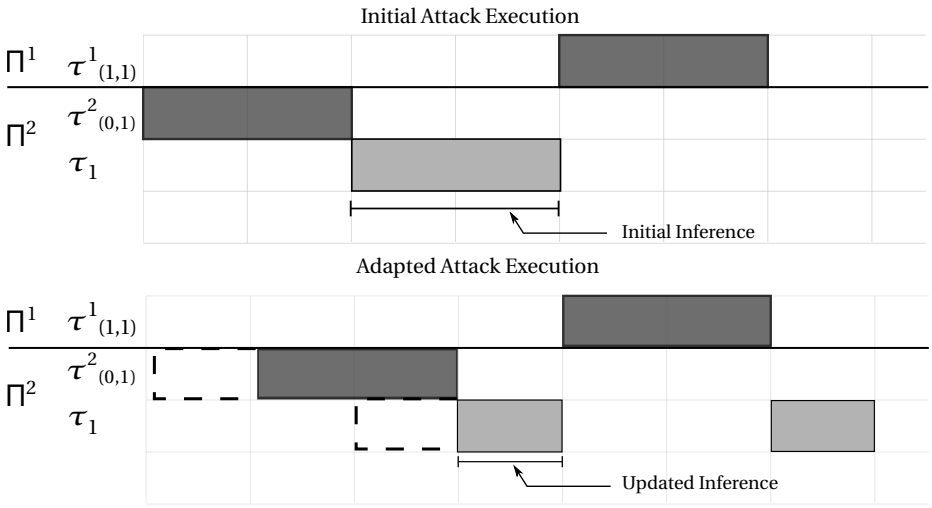


Figure 3.6: Delayed Execution of informant $\tau^2_{(0,1)}$.

3.2.4.3 Shortened Execution

Section 3.2.3.3 describes the two cases of preempted informant tasks. In both cases, shortened execution enables NOSYNEIGHBOR to find a more precise starting point of the respective victim node. From the constructed interval tree and interval data in Example III.1, the interval $[78, 90]$ has a length of 12. However, the task's allowed execution time is 6. Hence, the informant was preempted. According to the preliminary inference strategy (Section 3.2.3.3), this interval will be treated as a victim node and will be added to V. To find out the precise starting point of the preempting task, NOSYNEIGHBOR will instruct the informant to reduce its execution time by 1 unit in subsequent iterations until the requested execution time matches the actual execution time.

Figure 3.7 shows the schedule diagram of $[78, 90]$. In this example, $\tau^2_{0,0}$ was preempted by the victim task, and the shortened execution strategy placed the informant immediately before the victim task's execution. Such placement enables NOSYNEIGHBOR to escalate the passive attack into an active *anterior attack* [15]. Interestingly, iteratively

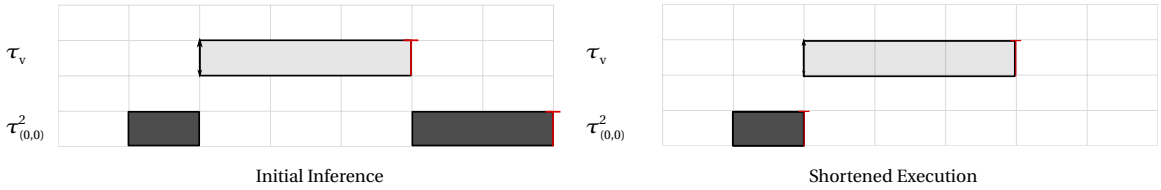


Figure 3.7: Shortened execution of informant $\tau^2_{(0,0)}$.

shortening the execution time also reveals the entire interval of τ_v, which can be calculated as $\tau_v = [I^2_{0,0,n}.end, I^2_{0,0,n-1}.end - 1]$, if the execution time is shortened by one unit in each iteration.

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

3.3 Evaluation

I evaluated NOSYNEIGHBOR on synthetic task sets to validate our approach and simulate the impact of NOSYNEIGHBOR on a range of realistic real-time systems. The evaluation shows the effectiveness of NOSYNEIGHBOR in detecting the execution of a victim task. Section 3.3.1 details the experimental setup used for the evaluation of NOSYNEIGHBOR based on the metrics discussed 3.3.2, and present our results in Section 3.3.3. Note that the following evaluation assumes that the victim task has the highest task priority in the system. The assessment results would vary based on the priority of the victim task

relative to the priority of the other functions in the system, including that of the informants. Since the main focus of this dissertation is to introduce the attack technique and the dependency of the attack on the informant parameters, the evaluation of the attack impact based on the victim's priority and the scheduling algorithm used are out of scope for this paper and would be an essential future work.

3.3.1 Experimental Setup

UUniFast [43] is a popular algorithm for generating synthetic tasksets for real-time systems. However, UUniFast can only be used for non-hierarchical systems. So, I extended the UUniFast algorithm to accommodate hierarchical systems. I term the extended algorithm as **iterated UUniFast** algorithm.

The iterated UUniFast algorithm first generates a synthetic set of partitions. Subsequently, each partition generates a set of tasks with random parameters, ensuring that all task sets are schedulable. This means that tasks can be successfully executed without missing any deadlines.

In the following experiments, we set the value of the major frame M as 1000. The period of each partition is randomly selected from the set of factors of M. Hence, M is divisible by $T^k \forall k$. Each partition has a total utilization $U^k = \frac{C^k}{T^k}$, which is then chosen by the UUniFast algorithm with a target utilization of 60% unless otherwise specified.

The budget of each partition is $U^k \times T^k$. Using the budget as the maximum possible execution of all tasks combined, we used UUnifast to derive the utilization vector for the partition's tasks with periods randomly picked from the set of factors of T^k. Similar to the

partition, we used each task's period and utilization to assign a budget. The priorities of all the tasks within each partition are set using rate monotonic scheduling [44].

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

Using these generated hierarchical tasks, we developed a *discrete event simulator*. In each iteration, we use K event queues, one for each partition, to select the highest priority task among all the partition queues and simulate execution. At each clock tick, all these executions are recorded in a *timeline*.

3.3.2 Evaluation Metrics

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

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

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

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

A detected window is considered a *True Positive* if the detected window is within the attack effective window (AEW) of the secure task phase. We identify true positives by

taking the intersection of NOSYNEIGHBOR's generated inference tree and an interval tree constructed from the actual execution intervals of the victim task. If a detected window is not in the set of actual execution windows, it is considered a *False Positive*. Here, precision is a proxy for the likelihood that an execution interval identified as being the victim's is, in fact, the victim's. We first find the intersection of the inference tree and the original execution times of the trusted task and divide it by the total number of reported victim inferences in ζ'.

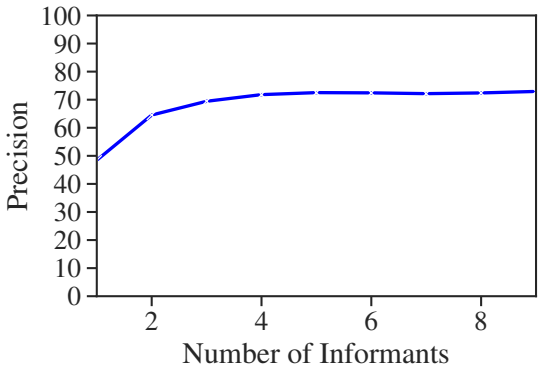
Recall: Recall shows the percentage of the correctly identified victim inference windows out of all the actual victim execution windows. This metric helps understand the quality of the predicted inference tree. We measure the recall using the following definition:

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

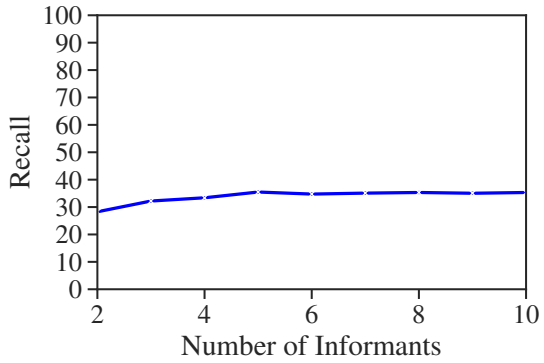
In III.5, the *False Negative* denotes the execution windows not successfully detected by NOSYNEIGHBOR. Note that a low recall indicates missed opportunities, while a high recall indicates the attack finds most of the victim's execution.

3.3.3 Experiments and Results

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



(a) Precision improves with the increasing number of informants



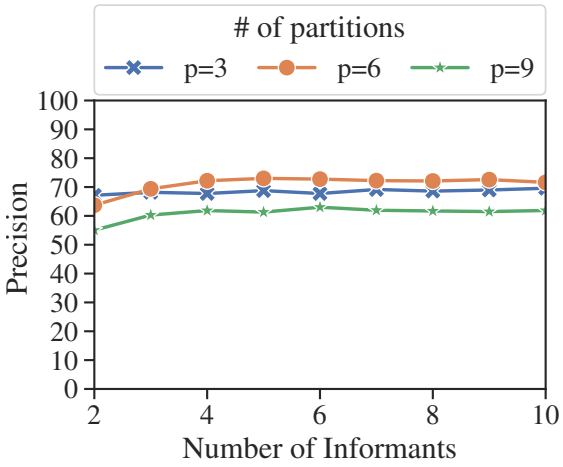
(b) Recall initially improves with the number of informants

Figure 3.8: Impact of the number of informants on the precision(3.8a) and recall (3.8b). Here, the system utilization is set to 60%, where 6 partitions have 4 tasks each, and only 3 partitions run malicious tasks randomly assigned to them.

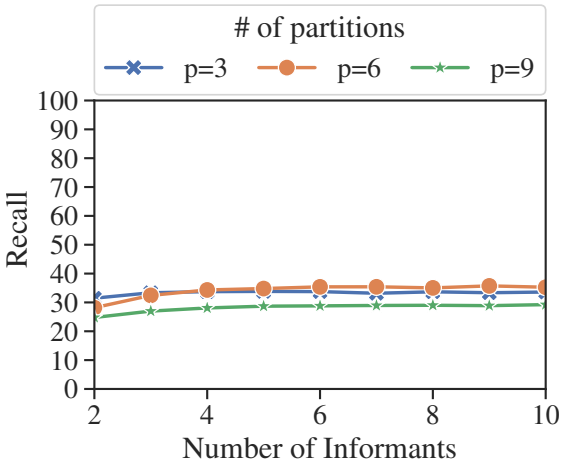
We evaluate the performance based on attack parameters in the second set of experiments. Note that the adversary does not decide NOSYNEIGHBOR's attack parameters used in the experiments; instead, these parameters are linked to an existing system's vulnerability that the attacker exploits.

Impact of Informant Count: We first evaluate the impact of the number of informants on the inference quality. We run NOSYNEIGHBOR on $10,000$ random task sets generated using the setup described in section 3.2. The system utilization is set to 60%, with 6 partitions, out of which 3 are impacted. Each partition has 4 tasks. The informant tasks are randomly selected from the affected partitions. Figure 3.8a shows no substantial impact on the number of informants on the inference precision with the given system parameters.

The recall also shows a similar trend in Figure 3.8b and shows minimal variation with the increasing number of informants. This is due to the adaptive behavior of NOSYNEIGHBOR that enables it to gain a similar level of precision even with a small number of



(a) The precision improves when $p = 6$, but goes down when there are more number of partitions further

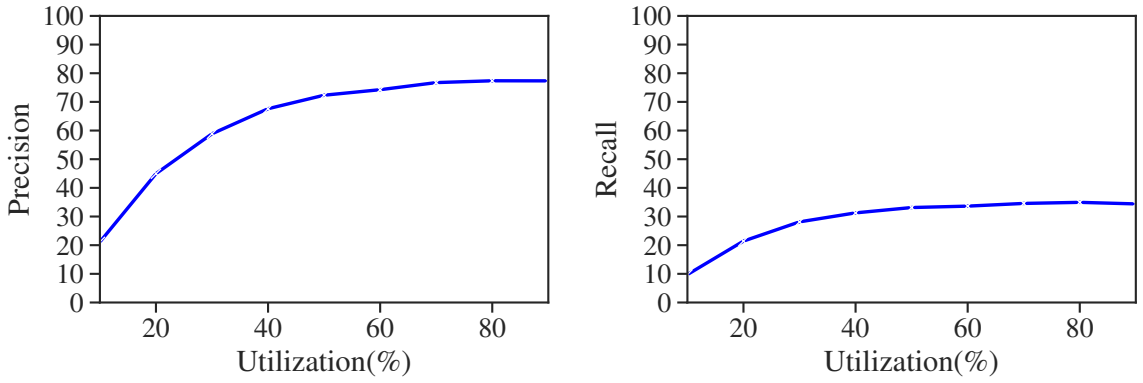


(b) The recall improves when $p = 6$, but goes down when there are more number of partitions further

Figure 3.9: Impact of the total number of partitions in the system. Here, the system utilization is set to 60%. Each partition has 4 tasks, and the number of partitions and informants are varied.

tasks. This result shows that NOSYNEIGHBOR does not need control over many tasks to determine an effective window for a successful attack.

Impact of Number of Partitions: We further extended this experiment in Figure 3.9a and Figure 3.9b, which shows that increasing the total number of partitions in the system does not significantly impact NOSYNEIGHBOR's performance. A higher number of partitions in the system adds more randomization when selecting affected partitions. Hence, the performance analysis observed in Figures 3.9a and 3.9b show that the performance of NOSYNEIGHBOR does not depend on the combination of partitions where NOSYNEIGHBOR executes. An adversary can leak the victim execution intervals with a precision of over 60% irrespective of the number and placement of affected partitions or informant tasks in the system.



(a) Precision increases with the total system utilization

(b) Recall increases with the total system utilization

Figure 3.10: Impact of utilization on precision and recall. The results are obtained with 4 informants, 6 partitions with 3 affected partitions, and the AEW is set to 500ms

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

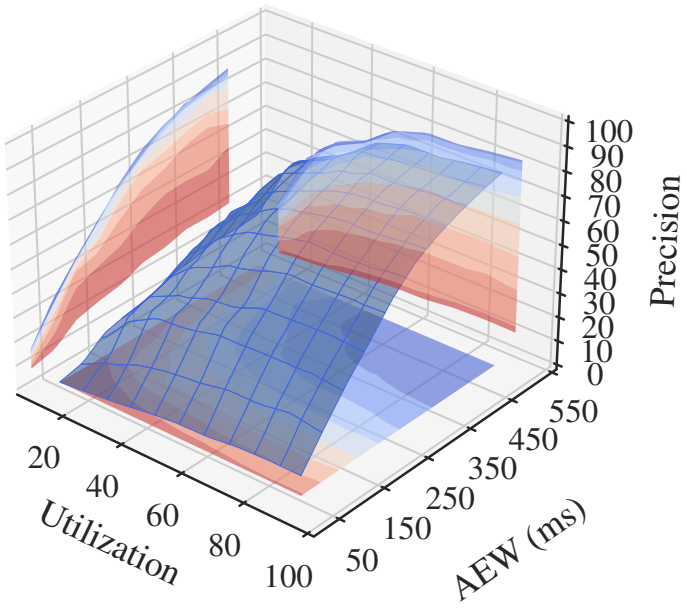
Figure 3.10a and Figure 3.10b show the result of this experiment. Notice that the precision and recall improve with increasing utilization. We observed that at lower utilization, the informants exhaust their budgets or meet their deadlines during parameter adaptation. However, they cannot find precise windows of the victim due to the lower utilization of the system. The derived windows in lower utilization levels are wide and outside the

AEW range. In the case of higher system utilization, where the schedule is *denser*, NOSY-NEIGHBOR can narrow the inference window by adapting the execution parameters. With a system utilization of 60%, the average precision is over 70%. 60% utilization is realistic for many real-time systems, and this experiment shows that NOSYNEIGHBOR can effectively infer victim execution under normal system loads.

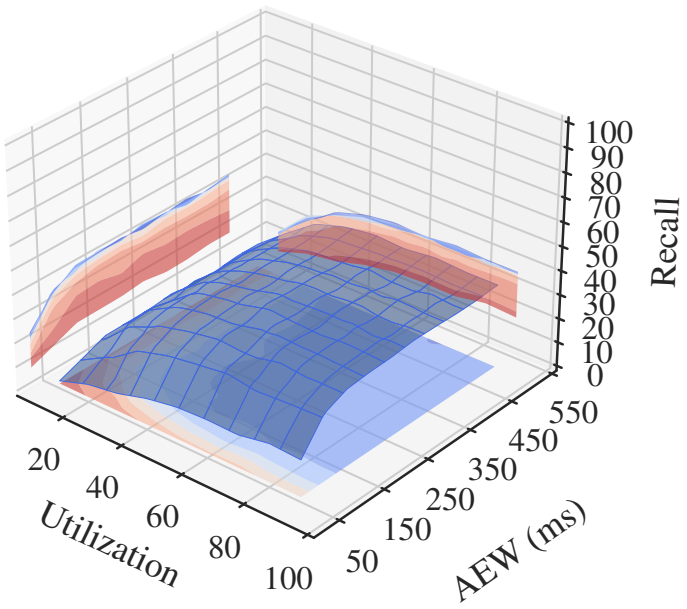
Attack Effectiveness: We combine the results from all previous experiments to analyze the effectiveness of NOSYNEIGHBOR inference. As delineated in Section 3.2, we employ AEW to refine the range of intervals considered as True Positives. For example, if the actual victim execution window is $[10, 15)$ and the AEW is set at 10, a detection window qualifies as a True Positive only if its range is a subset of $[0, 25)$. The integration of AEW enables the evaluation of inference performance through binary classification metrics; thus, detection windows can be labeled with True/False Positives and True/False Negatives, akin to traditional binary classifiers.

In Figure 3.11a, we observe that AEW can significantly impact the inference precision. This implies that for systems with very tight AEW, NOSYNEIGHBOR can have many false positives, which lowers the precision score of NOSYNEIGHBOR. However, it can still reach a precision of 30% at 60% utilization and AEW of 50 ms.

We observe in Figure 3.11b that the recall value is around 30% even for a wide AEW. This implies that NOSYNEIGHBOR is unable to identify all the execution windows of the victim task. This observation is the result of tasks with priority higher than NOSY-NEIGHBOR and corresponding informants. However, even if a small subset of the attack is successful, a safety-critical system can have a catastrophic impact.



(a) The impact of the size of AEW on the precision of the inference



(b) The impact of the size of AEW on the recall of the inference

Figure 3.11: Higher AEW of the victim allows better precision and recall under constant system utilization. Here, 3 out of 6 partitions are affected, with 4 informant tasks randomly assigned to the affected partitions.

3.3.3.1 Relationship of Precision and Recall in Attack Evaluation

In the above experiments, we used precision and recall to evaluate the effectiveness of NOSYNEIGHBOR. These metrics are used mainly for the evaluation of binary classifiers. However, in the context of detection windows, time intervals cannot be strictly classified as binary True Positives or False Positives. For instance, if a detection window encompasses the entire hyperperiod, one would attain 100% recall and precision, as all victim execution windows would be contained within the detection windows. However, such an approach is ineffective for targeted attacks and resembles a brute-force attack.

Typically, precision and recall exhibit an inverse relationship; that is, an increase in precision generally corresponds to a decrease in recall, and vice versa. This relationship is often referred to as the precision-recall tradeoff and has been extensively explored in the literature to derive its characteristics [45]. Nevertheless, the plots presented in Section 3.3.3 show a deviation from this conventional inverse relationship, indicating simultaneous growth in both recall and precision. This phenomenon can be attributed to the multi-stage enhancement of inference. Since only the final results of the multi-stage inference are graphed rather than each individual stage, the outcome reflects a higher count of true positives than observed in the baseline inference. This simultaneous growth of precision and recall aligns with prior studies indicating that a multi-stage detection system is the only condition to allow simultaneous improvement of precision and recall [46]. Such findings further substantiate the assertion that inference performance has been improved through the multi-stage adaptive inference of NOSYNEIGHBOR.

Despite the simultaneous growth of precision and recall, multiple system factors ultimately influence the final outcomes, and the underlying tradeoff persists; hence, an improvement in precision is likely to be accompanied by a reduction in recall. This phenomenon is evident in Figure 3.8a, where precision is significantly higher than recall, and any further adjustments to the algorithm or system parameters aimed at enhancing precision would likely compromise recall. Notably, at the saturation point indicated in Figure 3.8a, the values for precision and recall are complementary, with values approximately equal to 70% and 30%, respectively.

3.3.4 Case Study: NOSYNEIGHBOR Peeks Through the Blinds

We conducted a proof-of-concept NOSYNEIGHBOR attack on Litmus-RT. We used the P-RES [37] scheduler, which is a partition-based scheduler widely used in the literature for testing and validation of partition-based schedulers. In Blinder [3], the state-of-the-art randomization-based defense in hierarchical systems, the P-RES scheduler is extended to prevent timing side-channels by avoiding interference between tasks. The extended scheduler implementation is termed P-RES-NI. The P-RES-NI scheduler prevents task interference using a calculated priority inversion of higher-priority tasks.

Figure 3.12 shows the execution trace of NOSYNEIGHBOR adaptive attack on a sample task set under the default P-RES scheduler. Tasks τ_0^1, τ_1^1, and τ_0^2 are the informant tasks in the system, sending their execution timestamps to the NOSYNEIGHBOR task τ_1^2. The baseline inference derived by NOSYNEIGHBOR in the given execution trace shows that the victim task executed during the time windows $[81ms, 91ms]$ and $[99ms, 105ms]$. From the task set, we observed that the derived baseline inference is correct. NOSYNEIGHBOR will

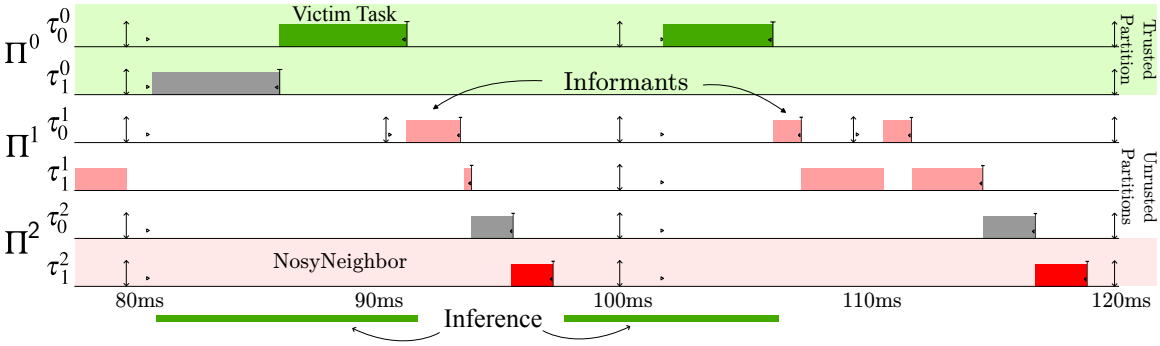


Figure 3.12: Executing NOSYNEIGHBOR in Litmus-RT under P-RES scheduler. The execution trace shows the inference drawn by NOSYNEIGHBOR using the execution timestamps of the informants τ_0^1 and τ_1^1.

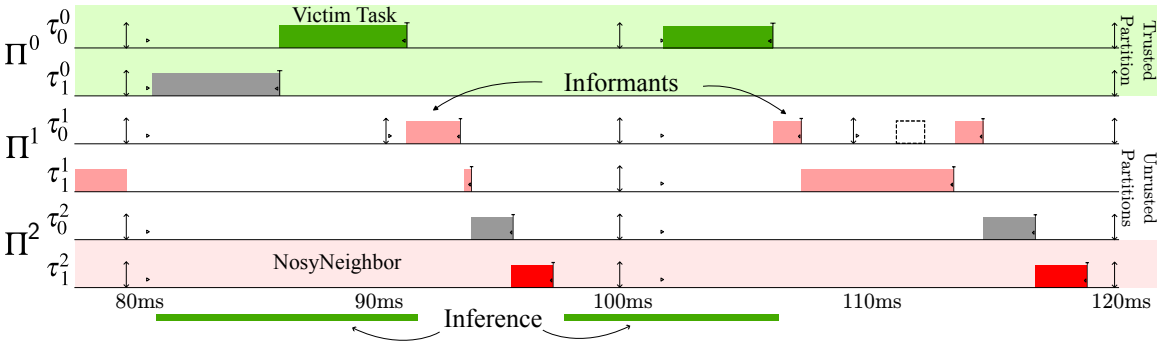


Figure 3.13: Executing NOSYNEIGHBOR in the presence of randomization-based defense with P-RES-NI scheduler

further improve the length of the inference window in the subsequent stages by adapting the parameters of τ_0^1 and τ_1^1 using the flow chart shown in Figure 3.4.

To prevent this kind of coordination of malicious tasks, Blinder shifts executing a higher priority task to prevent forming interference patterns on the lower priority tasks. In Figure 3.13, we implemented the P-RES-NI scheduler from Blinder and executed NOSYNEIGHBOR on the same task set as in Figure 3.12.

We observed two limitations of Blinder in our task model that make Blinder vulnerable to adaptive attacks. First, in Figure 3.14, we observed that Blinder is ineffective due to the tight deadline. The tighter deadline of task τ_1^1 at 100ms does not allow enough window

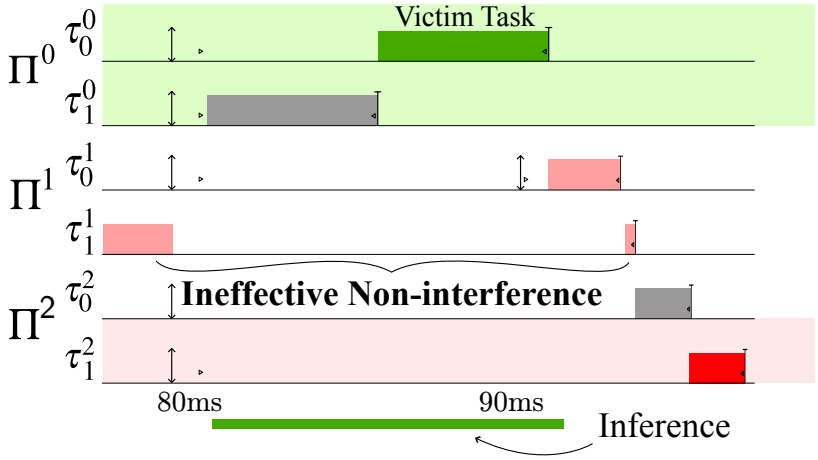


Figure 3.14: Zoomed-in view of Figure 3.13 from timestamp $80ms$ to $100ms$. The non-interference of Blinder is ineffective due to the constricted budget of τ_1^1

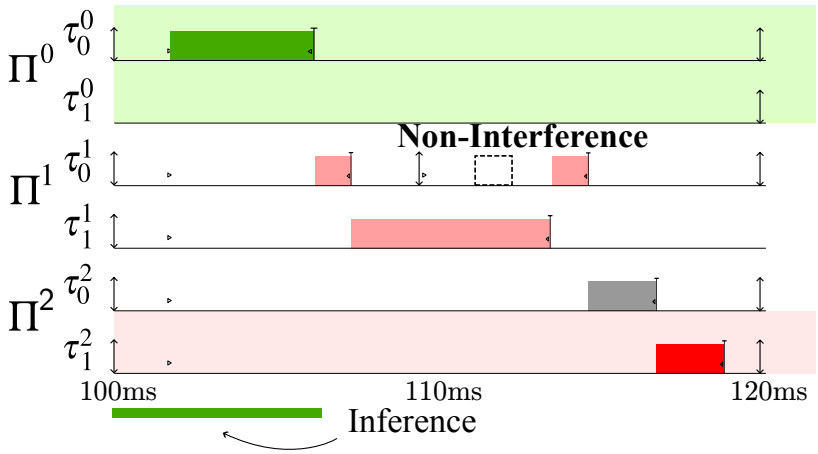


Figure 3.15: Zoomed-in view of Figure 3.13 from timestamp $100ms$ to $120ms$. Blinder is effective in creating non-interference. However, due to the collusion of the informants τ_0^1 and τ_1^1, NOSYNEIGHBOR could still make precise inferences about the victim.

for priority inversion without forcing the other tasks to miss the deadline. As a result, in Figure 3.14, the informants could still create interference patterns. NOSYNEIGHBOR uses Algorithm 2 to derive the inference window shown in Figure 3.14. Note that the end of the inference window precisely coincided with the ending of the victim task τ_0^0's. Hence, an adversary can use the given inference window to successfully execute an anterior attack [15] on the victim task even under the presence of randomization-based defense.

The second limitation of Blinder is highlighted in Figure 3.15. We observed that the P-RES-NI scheduler successfully prevented the formation of interference patterns through calculated priority inversion. However, when NOSYNEIGHBOR receives the execution time stamps of τ_0^1 and τ_1^1 at 115ms, NOSYNEIGHBOR uses Algorithm 1 to derive the baseline inference by combining the timestamps of the informant tasks to generate the victim's timing inference without using the preemption patterns between tasks.

3.4 Summary

This chapter presents a novel side-channel attack, termed NOSYNEIGHBOR, targeting hierarchical real-time systems. The effectiveness of this attack has been demonstrated against state-of-the-art schedule randomization techniques. The attack model employs adaptive task execution and utilizes common inter-partition communication channels, such as shared memory, to combine attack inferences derived from multiple malicious tasks within the system. Simulated evaluations of NOSYNEIGHBOR reveal a precision exceeding 70% under typical system loads. This attack underscores the shortcomings of randomization-based defense used in the state-of-the-art. It motivates further research into defense mechanisms against adaptive threats like NOSYNEIGHBOR that can circumvent schedule randomization.

CHAPTER IV

IMPROVING SOFTWARE SECURITY THROUGH MODULARIZATION OF LEGACY COMPONENTS[1]

Cyberattacks similar to NOSYNEIGHBOR (Chapter 3.2) can be propagated by exploiting vulnerabilities within source code. Such exploitation becomes increasingly feasible for attackers when the codebase has not been updated for an extended period and they have had sufficient time to analyze the vulnerabilities and develop corresponding exploits. For this discussion, I will refer to software components in the codebase that no longer receive official support from their vendors as *legacy* components.

The critical infrastructure sector, including energy, transportation, and defense systems, uses a substantial amount of legacy software due to the challenges of updating these components without disrupting their functioning. In this chapter, I present the use of modularization as a technique for facilitating software updates in critical infrastructure codebases.

[1]CHAPTER PUBLISHED IN [47]

I looked into the energy sector and studied the Experimental Physics and Industrial Control System (EPICS), which is an open source scientific cyberinfrastructure that is used in particle physics research and development. Specifically, EPICS enables the creation of distributed real-time control systems for scientific instruments such as particle accelerators, telescopes, and other large experiments. As in other ICSs, secure communication between nodes is essential to EPICS's overall security posture. EPICS depends on the networking implementations provided by the OS. One of the OSs EPICS uses is the Real-Time Executive for Multiprocessor Systems (RTEMS) [48].

Traditionally, the network stack implementation is a part of an OS that handles the networking tasks and the respective drivers for a network interface controller (NIC). The TCP/IP stack implementation of RTEMS historically also resided in the kernel and the user-level API declarations that RTEMS provides through the *Newlib* C library.

The *legacy stack* in RTEMS encounters several challenges. Firstly, this legacy stack lacks many features now considered fundamental in modern networking. Specifically, it is built on an older and less secure version of the IPv4, which is inadequate for many of the systems supported by RTEMS. Updating the network stack is challenging since it is tightly integrated with the RTEMS kernel. Consequently, any update to the network stack necessitates updating the entire RTEMS kernel. Unlike general-purpose systems, where updating software components can be straightforward, modifying any component within an RTOS requires that the new or updated elements adhere to the same timing constraints as the rest of the system.

To maintain its hard real-time nature, RTEMS does not have a separation between userspace and kernel space. All application and kernel programs run on the same priv-

ileged space to avoid the overhead of switching between them. This lack of separation makes RTEMS more vulnerable to cyberattacks. Through the modularization approach, the total codebase inside core RTEMS is reduced by over $270,000$ software lines of code (SLOC), which significantly reduces the trusted computing base of RTEMS and improves the security posture of RTEMS by allowing the core kernel components to be developed and tested separately from the communication module like the networking stack.

In this work, I separate the legacy stack into its own module outside RTEMS to facilitate switching the network stack without requiring heavy changes to user applications or the RTEMS kernel. This modularization uses extant Newlib header files to allow RTEMS users to build and link their applications to their network stack of choice, much like they can select among several different scheduling algorithms depending on the application needs [49]. This modular *Networking-as-a-Library* framework also allows users to build their own implementation of a network stack for RTEMS instead of depending on the legacy implementation provided by the OS. This approach provides two major benefits to EPICS. First, applications are not restricted in choosing networking features because they can link to a different network library without changing much of the code. Second, the network library provides an opportunity to upgrade to more secure, modern network stacks.

The process of modularization itself does not enhance RTEMS's security posture, but it can expedite research and development efforts related to the security of network stacks. By relocating the modularized codebase outside the real-time kernel, security researchers can efficiently patch known vulnerabilities and report newly identified issues without the constraints imposed by latency-sensitive kernel code. This strategy simplifies the skill set

required for system engineers, enabling them to concentrate exclusively on networking code and allowing for a more focused analysis of their modifications. Concurrently, kernel developers can devote their attention to core kernel components, unburdened by security concerns in communication channels, effectively treating the network stack as a black box for impact assessment.

To the best of our knowledge, these contributions make RTEMS the first monolithic RTOS that allows flexibility in choosing among multiple networking stacks. This flexibility makes RTEMS is one of the most adaptable RTOS for real-time system developers by providing them the option to select the network features that are specifically targeted toward their needs, hence enabling tradeoffs in performance (memory consumption, bandwidth, latency) and security.

4.1 Background

RTEMS is an open-source real-time OS. As such, systems built using RTEMS have temporal and logical correctness requirements. In addition, because the RTOS supports various size target platforms across different architectures (e.g., ARM, Motorola 6800, and SPARC), developers have endeavored to use code that can suit embedded and resource-constrained devices. In the early years of RTEMS, such code has been ported from the lightweight C library Newlib [50], which is another open source project focused on providing POSIX-compliant cross-compiled software that is widely used in embedded system projects. Specifically, RTEMS adopted several enhancements provided by Newlib (e.g., floating point support and math library [51]), including header files [52] for which RTEMS

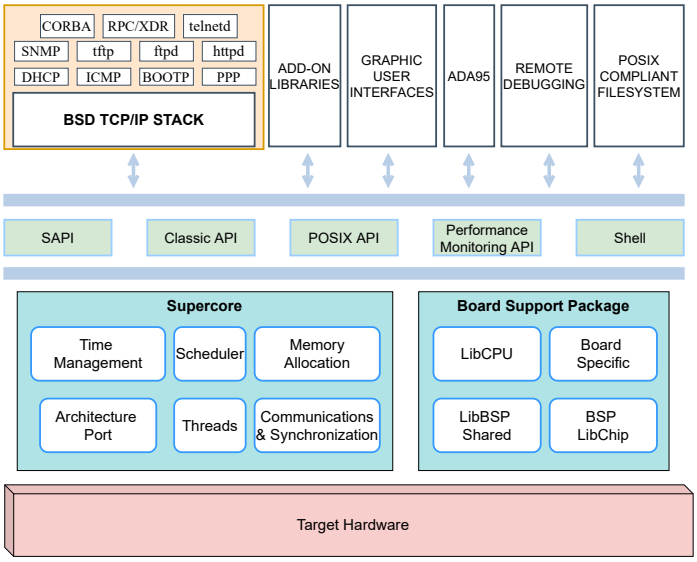


Figure 4.1: RTEMS High Level Architecture [1]

added its implementation. Some of these header files were used as a foundation for porting RTEMS network stacks. I provide more explanation regarding this in Section 4.3.

The RTEMS kernel (depicted in Figure 4.1) is composed of four main blocks: the Supercore, the board support package (BSP), the application programming interface (API), and the Services. The Supercore is the heart of the kernel. It provides the OS with real-time functionalities. As the name indicates, the BSP is responsible for providing all the necessary support to integrate the different hardware targets RTEMS supports. The API block allows RTEMS users to access the functionalities of the Supercore. RTEMS user-level services range from enabling programming in multiple languages to accessing additional libraries, including the newly added RTEMS network stack.

RTEMS provides a legacy implementation that is built into the RTEMS kernel. This legacy stack was ported from earlier versions of FreeBSD and has been part of RTEMS since the late 1990s. FreeBSD integrated DARPA's TCP/IP stack [53] in its early network stack implementation. The stack generically regroups the OSI communication model into

four layers: the application layer (which combines the session presentation and application layers of the OSI reference model) transmits user application data to the transport layer using the sockets API. The transport layer uses TCP and UDP protocols over the IP (or Internet) layer. The Network Access layer (which integrates the OSI model's data link and physical layers) is the network stack's lowest layer. It handles the physical hardware and protocols required to deliver the data across a physical network. This handling is done through the device drivers in RTEMS, which are responsible for initializing and operating the embedded hardware's NIC.

A network application uses the networking APIs, like the socket API, to make system calls with the appropriate protocol headers, which triggers the network drivers to send physical signals to the hardware to carry out the requested action. Traditionally, the networking implementation is a part of an OS that handles the networking tasks and the respective drivers for a NIC. The implementation also provides user-level header files that contain the declarations for the user APIs. In RTEMS the POSIX networking API signatures are provided to the applications through the *Newlib* C library, and the implementation of the TCP/IP stack along with the NIC drivers, were a part of the RTEMS legacy stack.Although multiple targets have used the RTEMS legacy stack for a long time, it did not evolve at par with the developments in the FreeBSD stack due to the following reasons: first, making changes inside the kernel requires significant time and expertise. Next, making any change to the legacy network stack was essentially a change to the RTEMS kernel, which involves a lot of regression testing.

In addition to Newlib, RTEMS uses FreeBSD's code base, similar to several other well-known OSs. FreeBSD [54] is also another open-source OS known for its high per-

formance in modern systems. An RTEMS repository named *rtems-libbsd*, or the libBSD module, was built by RTEMS developers to port the required codes from FreeBSD, which also includes the API implementation for the Newlib header files. The libBSD module uses a git submodule to track the upstream FreeBSD source code. LibBSD uses Python scripts to port specific files from this FreeBSD submodule as follows: first, a block of FreeBSD source code is imported from the submodule. Then, the necessary files are copied locally to the RTEMS-libBSD repository and adapted to work with RTEMS through the scripts which not only imports the code but also adds RTEMS-specific header files to them to properly connect the FreeBSD drivers to the RTEMS kernel. (I refer to this approach as the libBSD framework in the remainder of the paper.)

In recent years, the RTEMS developers have used the libBSD framework to import FreeBSD's TCP/IP stack, providing users with the option to use a modern and secure FreeBSD network stack with their RTEMS applications. The LibBSD, which uses the FreeBSD network stack, has a complete IPv6 support along with robust security features [55]. The modern features present the libBSD stack as a great upgrade option to a more modern stack. One caveat to having libBSD as the only alternative to the legacy stack is that some targets have very limited available memory and are incapable of running the libBSD stack. Thus, I prepared an adapter version of the lwIP [56] stack as an alternative to the libBSD. lwIP is an independent project targeted towards embedded systems with strict memory constraints. The features of the lwIP stack are comparable to that of libBSD, but size and memory requirements are much smaller than that of an application linked with libBSD. Some of the essential highlights of the lwIP stack are the much-required IPv6 sup-

port and the support for IPSEC, which has been studied and added by other independent projects [57].

In the following section, I present a modular network stack approach that decreases the reliance on the legacy stack and provides additional network stack options. This approach also allows RTEMS users to develop more suitable network stacks without modifying the entire RTEMS kernel.

4.2 Motivation

A modular network stack approach has been previously attempted on microkernel OSs like HelenOS [58] where each part of the network stack works as a server module for the microkernel proof-of-concept implementation. In contrast, our work is based on a Monolithic Real-Time kernel, where I have implemented the whole network stack as a separate library module that gets linked into one whole executable binary, which is run on the target hardware.

NetBSD also uses a modular TCP/IP stack implementation through a rump Kernel TCP/IP [59] that virtualizes kernel functional units into clients. The clients can be one of three types: local, microkernel or remote. The local client-type approach uses rump kernel as a library with rump API calls. Instead of adding a new API layer, our approach provides support for the common API calls for multiple stacks and an application does not require any change in terms of includes API calls for working with an RTEMS networking library.

I have extended our unique approach to add an independent networking stack lwIP, which has been used in RTOSs before [60, 61], but our work differs in two ways. First,

the Independent networking implementation has not been integrated into the kernel, in contrast to the FreeRTOS TCP/IP implementation, which is part of the kernel. Second, the networking module provides a framework for adding and modifying any layer of the network stack without affecting the main kernel, which will pave the way for support on a wider range of architectures and NICs.

4.3 Modular Network Stacks

RTEMS users currently face the following challenges related to the implementation of the network stacks: (1) difficulty upgrading the legacy stack, (2) inability to fully utilize each of the existing network stacks (legacy and libBSD) because of a lack of appropriate drivers, (3) lack of security support in the legacy stack. To address the first challenge, I separated the components of the legacy stack from the current RTEMS kernel into its own standalone repository (see Section 4.3.1).

To resolve (2), I have separated the drivers from the RTEMS kernel and integrated them into the networking module. Additionally, I created a standalone submodule called *rtems-net-services*, which can be incorporated into any RTEMS network stack to provide networking services such as the File Transfer Protocol (FTP) and Trivial FTP (TFTP). These services are accessible by any network stack module (see Section 4.3.4). Furthermore, as part of our ongoing efforts, I have streamlined the workflow for adding support for specific hardware platforms to enhance compatibility across all network stacks. I demonstrated this new workflow and conducted experiments on an uCdimm ColdFire 5282 Microcontroller Unit (uC5282), a low-resource board with memory as low as 512kB. uC5282

is used in EPICS for RTEMS-based projects, and the modularization of the legacy network stack allows continued use of the device in production while appropriate alternatives are being explored.

To address the third challenge, I use a network stack implementation that provides modern security features such as IPv6. The lwIP network stack implementation [56] matches such a requirement. In addition to IPv6, lwIP can also be combined with other independent protocol implementations like embedded IPSec [62]. lwIP in combination with embedded IPSec has been evaluated with microkernel OS [57], showing that lwIP can be robust and versatile when adding security updates. Moreover, the lwIP network stack is targeted towards embedded systems with strict memory constraints. As such, I broadened the existing network stack options by fully integrating a third network stack module based on lwIP. The lwIP-based networking stack, called *rtems-lwip*, will enable the users to choose the network stack that provides the necessary security required for the application (see Section 4.3.3).

As a result, a new architecture is obtained for the network stack library, as shown in Figure 4.2. In the following subsections, I describe how the stacks and the net-services module were built to form the network stack library in further detail. Table 4.1 shows a comparison of the features of the network stacks.

Table 4.1: Comparing network stack features

Feature	**LibBSD**	**lwIP**	**legacy**
TCP	✓	✓	✓
UDP	✓	✓	✓
IPv4	✓	✓	✓
IPv6	✓	✓	×
IPSec	✓	✓	×

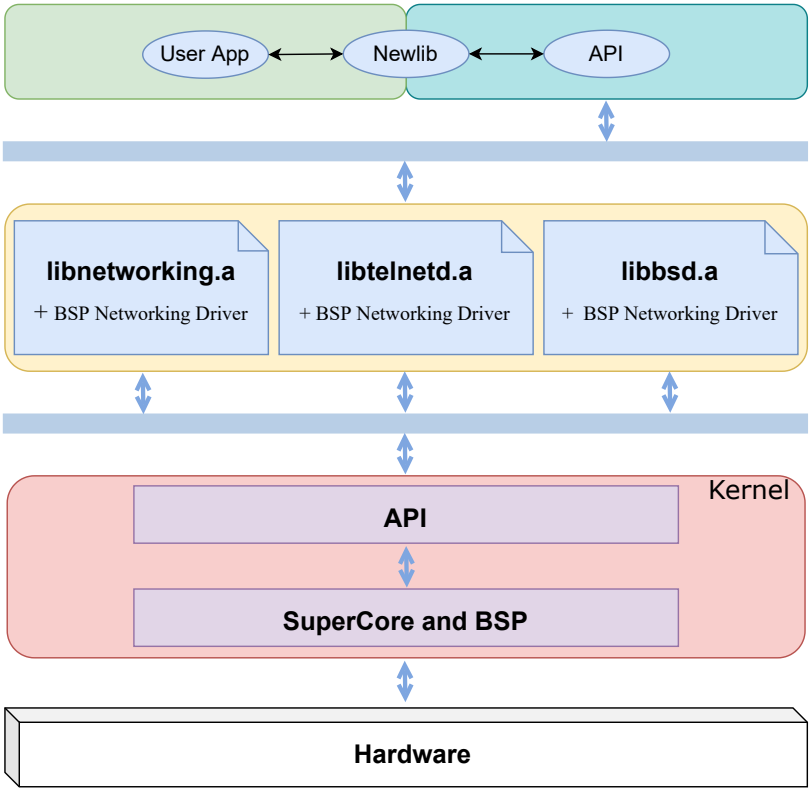


Figure 4.2: RTEMS with Modular Network Architecture [1]

4.3.1 Legacy Networking Module

Any significant modification to the legacy stack requires extensive changes to the kernel. Thus, to ensure that any update still satisfies and preserves the current networking functionalities of RTEMS and to make the legacy stack available to the projects that are actively using it (without any change in their project), I opted to separate the legacy stack from the core of the OS in the form of a static library (*libnetworking.a*). A static library allows us to treat the existing network stack as a separate unit in the OS without requiring the code to be built along with the OS kernel. Building the static library requires that we slightly modify the current flow of the RTEMS Networking, and link the *libnetworking.a* library to the user application directly. In the new separate legacy stack, I followed the same

directory structure that was maintained in RTEMS, to make it easier for interested developers to maintain this stack separately without having to adapt to a different organization of the same codebase.

The implementation of the legacy stack was located within the RTEMS *cpukit/* directory. In contrast, the BSP-specific drivers for legacy networking resided in the *bsps/* directory for each hardware target supported by RTEMS. To facilitate the integration process, I established the *rtems-net-legacy* repository [4], which contains the TCP/IP implementation files and BSP drivers. During compilation, user applications statically link to the *libnetworking.a* library, which encompasses the legacy TCP/IP implementation alongside the BSP networking drivers utilized by the legacy stack. To create a straightforward and easy-to-integrate system for developers, I chose the Waf build system [63] primarily because it is written in Python, a general-purpose scripting language. This decision allows developers to concentrate on writing functional code, thereby saving time on adjustments to the build system.

The new build process comprises three stages (Configuration, Build, Link) as shown in Figure 4.3. During the Configuration stage, Newlib provides the networking API header files the user application uses. Then a Waf script (*wscript*) collects the build context , which consists of the target name, toolchain executable locations, build flags, and other environment variables. In the Build phase, a script (*netlegacy.py*), which I added to the repository, uses this build context and collects the required files to build. The selection of files is important to ensure that the correct driver gets linked according to the build context. The linked driver connects to the RTEMS kernel through the *bsp.h* header file, which is a BSP-specific header file present in all the RTEMS BSPs. This header contains the macro

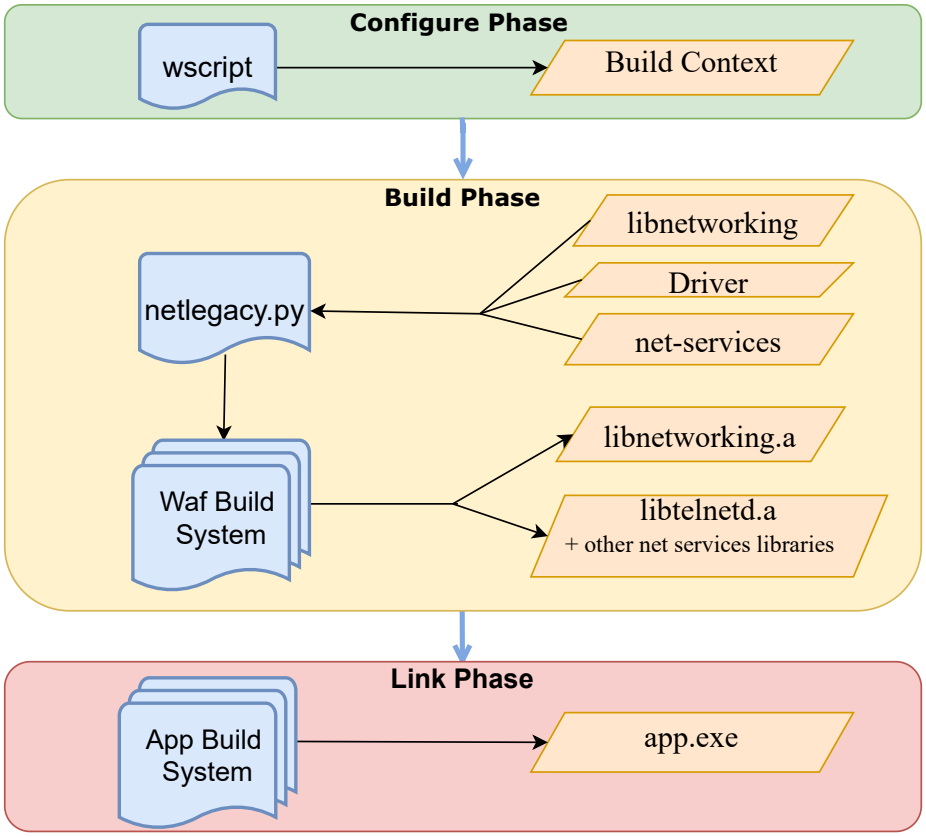


Figure 4.3: Modular Network Stacks Build Process [1]

defined for *RTEMS_BSP_NETWORK_DRIVER_ATTACH*, which declares the name of the driver attach function that the BSP will call to initialize the network interface. The driver attach function is defined in the *libnetworking.a* library generated from the waf build. This library is then linked to the user application in the Link phase. Since the end product from the Build phase is a separate C library, the user can use their build system to link to the library.

4.3.2 LibBSD Module

RTEMS uses this LibBSD framework to port the current TCP/IP network stack from the FreeBSD sources, which provides a full featured IPv6 [64] supported network stack.

To accommodate this BSD stack addition to RTEMS without changing the RTEMS source code significantly, the TCP/IP API header files required for the current BSD stack are pushed into upstream Newlib repository under the directory *libc/sys/rtems/include/*. Like the legacy networking process, the LibBSD builds a static library *libbsd.a* from the FreeBSD ported codes. This separate library is especially interesting for the network stack, as the networking API declarations are provided by Newlib while the implementation is obtained from libBSD. Using this approach, any application that makes use of the latest BSD networking can link to *libbsd.a*.

4.3.3 LiblwIP Module

The lwIP stack has been used in multiple embedded OS projects, such as FreeRTOS and HelenOS. In the RTEMS community, some users have also individually developed RTEMS drivers in order to support the lwIP TCP/IP stack for their projects. For example, the "uLan protocol for RS-485 9-bit network" project [65] has adapted the lwIP stack for RTEMS along with a driver for their target board ARM based TMS570. There are multiple such independent projects that are using the lwIP TCP/IP stack, but developments made on these projects are unavailable for an RTEMS user out of the box. To address the issue of scattered lwIP drivers, and to provide an alternative to the *libbsd* and *legacy* network stacks, I built a standalone networking module for RTEMS that can act as a centralized location for drivers, hardware abstraction layers (HAL), and adaptations developed by independent projects to use the lwIP stack with RTEMS.

The newly created *rtems-lwip* repository [5] also uses the Waf build system and has the same modular structure as the previous two stacks. I have also added the upstream *lwIP*

repository as a submodule to *rtems-lwip*. This submodule tracks the upstream changes, making it easier to update to the latest ones without starting a whole new self-hosted independent project. Along with our adaptation of the lwIP stack, I added network drivers developed by Texas Instruments for the ARM-based BeagleBone Black board to test the build process.

4.3.4 Net-Services Submodule

To further push for modular options, I moved some of the common net services (for example, *tftpfs* and *telnetd*) from the RTEMS kernel into a separate repository designated *rtems-net-services* that acts as a submodule to *rtems-net-legacy*. This new submodule builds static libraries (such as *libtftpfs.a* and *libtelnetd.a* for ftpfs and telnetd respectively) for the networking services.

The creation of the *rtems-net-services* submodule demonstrates that placing the RTEMS networking services in a module is both effective and maintains usability and function as it does not add any extra layer of build process for the user. Users who rely on the RTEMS networking services can build only the services they need, reducing executable size since these services are no longer in the kernel.

4.4 Evaluation

In this section, I present three evaluations of the RTEMS modular network stack framework presented in Section 4.3. In the first experiment (in Section 4.4.1), I show that

the modular networking framework does not require substantial effort from the user. To do so, I demonstrated the building of the *rtems-net-legacy* module using the *waf* system.

In the second experiment (Section 4.4.2), I analyzed the memory requirements to implement each of the three network stacks (legacy, libBSD, lwIP). Specifically, I computed and compared the size of the binaries of the same application over the legacy and libBSD network stacks. This experiment highlights the memory requirements between the stacks and shows the potential implications of switching from the legacy stack to the *rtems-libbsd* stack.

In the final experiment (Section 4.4.3), I evaluated the round trip times (RTT) of the *rtems-libbsd* and *rtems-net-legacy* stacks Although the lwIP network stack is fully integrated and adapted to RTEMS, the driver support is limited for an RTT analysis. Therefore, the lwIP stack was not used in this experiment.

For all three experiments, I have selected the uCdimm Coldfire 5282 (uC5282) as our hardware target due to its wide use in projects that deploy EPICS and RTEMS for safety-critical applications. The uC5282 microcontroller module uses the Motorola MCF5282 microcontroller that has an integrated 10/100 Fast Ethernet Card. The uCdimm platform has an onboard Synchronous Dynamic Random Access Memory of 16MB.

4.4.1 Building a Classic Application Using Net-Services

In this section, I illustrate the process of building a simple Round-Trip Time application using the *legacy* network stack. The application building follows a two-step process as described in section 4.3.1. The network stack is configured for the target hardware with the command shown in Figure 4.4, then built using the *'./waf'* command.

```
rtems-net-legacy $>./waf configure \
> --prefix=$RTEMS_PREFIX \
> --rtems-bsps=m68k/uC5282
```

Figure 4.4: Command to configure and build *rtems-net-legacy* stack for uC5282

For rapid functionality testing of the network stacks on uC5282, I have also provided QEMU emulator [66] support for the uC5282 target. The current main branch of QEMU does not have support for the target board so I refactored an old patch [67] to make the board compatible with the current QEMU. I will contribute this added support to QEMU upstream.

4.4.2 Size Comparison of Binary Images

To compare the sizes of the binary images of the three network stacks, I used the same RTT application that I built in the experiment in 4.4.1. I used the GNU *objcpy* and *size* tools from the GNU toolchain for the *m68k* target, to get the binary images of the executable linked to different stacks, along with the size of *text*, *data*, and *bss* segments to understand the memory usage of the apps in each network stack.

Table 4.2 shows the results from comparing the size of the generated binary images. The size difference between the *libbsd* stack and the other two stacks is significant. However, the size difference between the *lwIP* stack and *legacy* stack is much lower. The sizes of the *.text* segment (which in all three cases represents the largest portion of the executable) show that the libBSD brings in a much larger code, making the executable much larger compared to the other two. The *.data* segment shows a similar pattern where, interestingly, the lwIP stack has the lowest value. This low value is due to the optimized memory design

of the lwIP stack, which enables it to run on targets as low as 512kB of memory. The *.bss* segment in the lwIP stack can be reduced by allocating even lower memory to the lwIP configuration, which can be done in the *rtems-lwIP* repository through the *lwipopts.h* header. This similarity in the lwIP and legacy stack size shows that *rtems-lwip* can be a suitable alternative to the *legacy stack* for memory-constrained targets.

Table 4.2: Size comparison of binary images (all values in kB)

Network stack	**.text**	**.data**	**.bss**	**Total Size**
rtems-libbsd	1273	58.4	24	**1,332**
rtems-net-legacy	244.4	6	44	**250.5**
rtems-lwip	293	1.7	59	**294**

4.4.3 Round Trip Time Analysis

The RTT analysis shows the *latency* of the network, which gives an idea of how much time it takes for a packet to be transferred. A comparison of the RTT over the loopback device shows the latency from the network stacks only without other factors can affect the latency, like the wiring and routing overhead due to the connection between devices.

To compare the RTT, I created a lightweight application that sends a constant-size packet over the Internet Control Message Protocol (ICMP) using raw sockets. The ICMP header size is $28B$ and I added a *padding* buffer of $56B$ to send a total of $84B$. From the recorded data over 10 runs (see Figure 4.5), I noted that the LibBSD stack has a latency overhead of approximately double the average latency from the legacy stack. This observation shows that switching an application to the FreeBSD-based *libbsd* stack will have a performance overhead that can accumulate every time a packet is sent or received. This overhead might become critical in high-precision industrial controllers where the latency

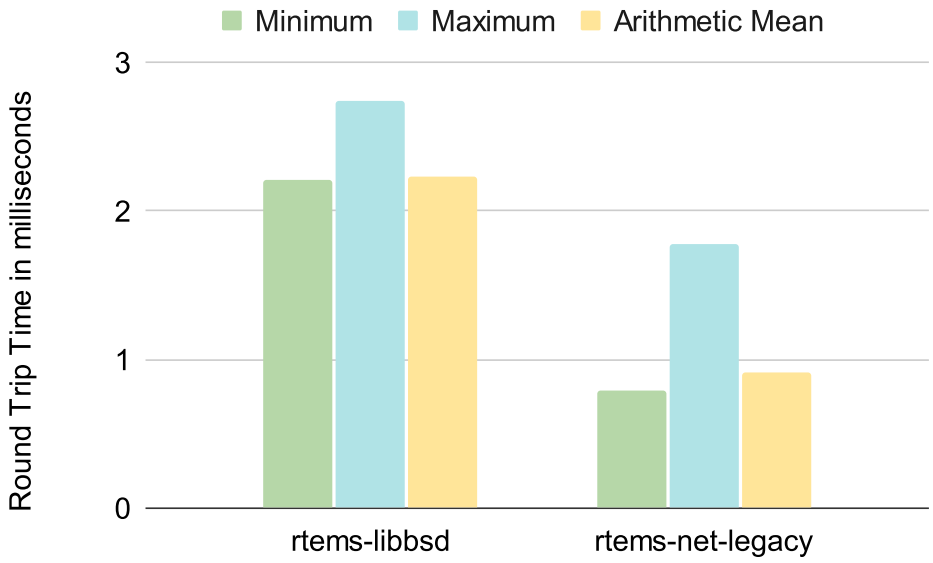


Figure 4.5: Round trip time comparison of the RTEMS network stacks [1]

of the network can impact the validity of observed values. The latency analysis reinforces the need for a lightweight network stack alternative, which will be available to the user through the rtems-lwip module.

4.5 Summary

This chapter explores the challenges of upgrading software components in real-time embedded systems, primarily due to the complexity of the codebase and the stringent timing constraints that each line of code must meet. It introduces the concept of codebase modularization within RTEMS, a monolithic RTOS, focusing on migrating the networking stack implementation outside the kernel and into a standalone repository. This approach, termed *Networking-as-a-Library*, is novel in a monolithic RTOS, as it effectively maintains the interaction between the kernel and the communication modules without forcing any specific implementation on RTEMS and allowing the user to choose from the network-

ing implementation based on their requirements. The findings from this research have been adopted by several national laboratories, enabling them to use legacy system components alongside the exploration of newer networking stacks without experiencing system down-time. The codebase and the developed networking modules as a part of this chapter have been open-sourced and freely available for use [4, 5].

CHAPTER V

REAL-TIME RECOVERY OF SYSTEMS UNDER ATTACK[1]

Increasing attacks on CPS have motivated research in CPS security. One such security approach is the Simplex architecture. Figure 5.1 depicts the architecture comprising three primary components: safety unit, complex unit, and decision module. The safety unit contains a *fully verified* controller that is outside the reach of an attacker. On the other hand, the complex controller makes significant use of commercial off-the-shelf (COTS) components that are not verified and can be vulnerable to attacks. The decision module is responsible for switching the operation mode between safety and complex controller to ensure the CPS plant is functional throughout the timeline. The use of the COTS component exposes the complex controller to known vulnerabilities. A restart-based approach has been previously used to strengthen the security of the complex controller [25, 68, 69].

This work presents a secure boot integrated restart-based approach that periodically restores the real-time complex controller into a secure computing environment. The secure boot mechanism prevents the installation of persistent rootkits or compromised OS images

[1]CHAPTER PUBLISHED IN [1]

from taking over a system. Though the secure boot sequence ensures a trusted computing environment after every restart, its use in safety-critical systems is limited due to the lack of thorough timing analysis for real-time systems.

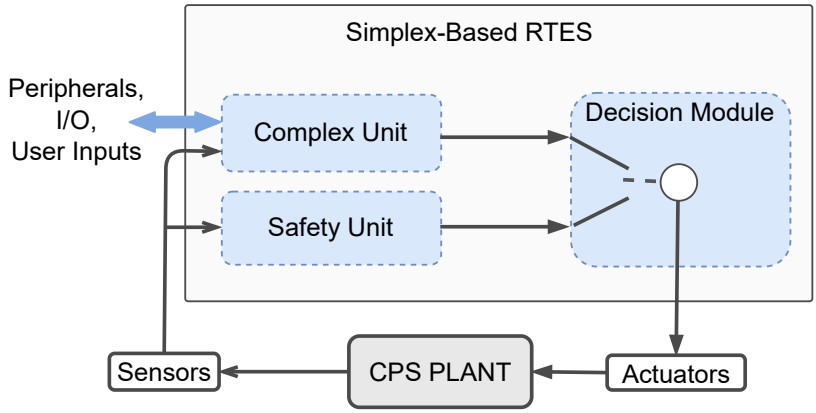


Figure 5.1: Architecture of CPS plant with Simplex-based RTES Controller [1]

5.1 Adversary Model

As shown in Figure 5.1, the RTES's complex controller unit can be accessed from the CPS network through the system's input interfaces and peripherals. The threat model described in Section 3.1.2 acts on this complex controller. I assume that the safety controller is located on a separate and isolated partition of the RTES and, therefore, cannot be accessed by the remote attacker. I also assume that all software components of the complex unit in the RTES are trustworthy initially, i.e., trusted system developers cryptographically sign their static images.

Consistent with previous studies [23], the goal of the attacker in this work is to control or tamper with the plant's operation. To achieve this goal, the attacker manipulates the real-time operating system (RTOS) image or real-time and control applications in the complex

controller. This manipulation impacts the integrity of the actuator commands computed by the complex controller.

5.1.1 Secure Boot-Enabled Simplex System

Our goal in this work is to integrate security functionalities to prevent the adversarial actions presented in Section 5.1 while attempting to minimize the impact on the system's performance. In this section, we present the design of the secure boot-enabled RTES and analyze its schedulability performance.

5.2 System Design

An RTES compromised by an adversary at runtime can be recovered by resetting the complex partition to its initial trusted state using an external timer input and a secure boot-enabled restart operation. The safety unit provides a mechanism to initiate the secure restart operation on the complex unit, and regardless of the partition's current state (secure or under attack), the safety unit sends a hardware pulse to the complex controller reboot pin. Hence, an adversary is unable to block the restart operation. The secure-boot-enabled restart ensures that (1) compromised software components are disabled and (2) only authenticated software components are activated in the complex unit once the partition is restored.

Both the deactivation (of possibly corrupted software) and authentication (of trustworthy software) are achieved through a signature verification mechanism started from the root of trust, a system component trusted for measurement and verification at all times.

The root of trust comes in the form of software and hardware components. A software root of trust can be stored in a secure read-only memory (ROM) location and is responsible for checking the signatures of subsequent components locally or with the help of trusted hardware. This work uses a bootloader as a software root of trust, a trusted component in traditional computing systems that initializes a system's software stack. A bootloader is also usually lightweight and suitable for resource-constrained platforms such as an RTES.

This signature verification approach guarantees a secure computing environment *only at restart*. That is, once the RTES has been securely rebooted, the adversary can once again attempt to modify the RTOS and other applications to regain control of the complex controller. Thus, to limit the potential impact of an attack, we routinely verify the authenticity of the software on the platform by performing a periodic secure reboot.

5.3 Schedulability Analysis

We now analyze the operation of the complex unit with periodic secure reboots enabled and derive the schedulability conditions for a secure-reboot system. First, we formally define the RTES tasks necessary for the analysis as follows: we consider that the complex controller is a uniprocessor system that executes a set $\mathcal{T}$ of periodic tasks using a fixed priority preemptive scheduling algorithm prior to integrating the periodic secure reboot functionality. Each periodic task $\tau_i \in \mathcal{T}$ is characterized by a tuple $\{c_i, T_i, i\}$, where c_i is the WCET of the task. T_i is the period, i.e., an instance of the task is periodically released at a regular interval of T_i units of time (we denote by $\tau_{i,m}$ the instance released at time mT_i). i is the task's priority. Priorities are assigned such that for two tasks τ_i and

τ_j, if $i < j$ then τ_i has higher priority than τ_j. In addition, we assume the tasks to have *relative deadlines*, that is, a task released at mT_i, for an arbitrary integer m, must complete its execution by $(m+1)T_i$ (we refer to the implicit deadline simply as deadline herein). We denote the total utilization of $\mathcal{T}$ by $U = \sum u_i$ where $u_i = c_i/T_i$.

The WCRT of τ_i on a unicore processor, with fixed priority preemptive scheduling [70, 71], can be calculated using the recurrence relation by Audsley et al. [72]:

$$R_i(n+1) = c_i + \sum_{i_j<i} \left\lceil \frac{R_i(n)}{T_j} \right\rceil c_j \tag{V.1}$$

where $R_i(n)$ is the value of the WCRT calculated at the nth step of the iteration. The equation terminates when $R_i(n+1) = R_i(n)$ or $R_i(n) > T_i$. The base condition for the recurrence relation can be taken as $R(0) = c_i$.

To integrate the periodic secure reboot, we model the reboot procedure as a periodic task τ_r with a WCET of c_r, a period of T_r, and a priority r. Since the reboot process is capable of preempting all ongoing processes in the complex controller, we consider that τ_r has the maximal priority r in the system, i.e., $r < i,\ \forall \tau_i \in \mathcal{T}$. Also, for the restart task, the WCET c_r can be viewed as the duration between triggering the reset pin of the controller to the instant the first task of $\mathcal{T}$ starts execution. This duration depends on the controller's mode of operation. For our analysis, we distinguish three modes: For a system with no restart task, we let $c_r = 0$. When a periodic non-secure restart is added, we can assign $c_r = \epsilon$, where ϵ represents the duration of the system restart. Finally, for a mode of operation that integrates the periodic secure reboot functionality, $c_r = \epsilon + \epsilon'$, where ϵ'

represents the overhead due to secure boot verifications. Since the restart procedure is the same for every restart, c_r is considered constant.

Besides c_r, another timing parameter we must study to perform an accurate schedulability analysis is the maximum number of restarts a task τ_i can be subject to before completing a single execution. The worst-case number of restarts can be characterized by first understanding the secure reboot mechanism: If a task is already executed when the reboot is triggered, it will be terminated and flushed along with the rest of the system memory. However, if the task has not been released yet, that task will be scheduled even if the task and restart are released simultaneously. Using this distinction, we formulate the following Lemma derived from Eq. V.1:

Lemma V.1. *WCRT for an arbitrary task in a secure-restart-based RTES is found when the following recurrence relation is satisfied:*

$$R_i(n+1) = c_i + c_r + \sum_{i_j<i} \left\lceil \frac{R_i(n)}{T_j} \right\rceil c_j \tag{V.2}$$

converges, i.e., $R_i = R_i(n+1) = R_i(n)$.

Proof. Let us draw a relation between an arbitrary task $\tau_i \in \mathcal{T}$ and the reboot task, τ_r for an arbitrary instance $\tau_{i,m}$. We know that the release time of $\tau_{i,m}$ is mT_i and the relative deadline is $(m+1)T_i$. Similarly, we can assume a reboot task with period T_r. The least common multiple of the periods of all tasks except the reboot task is a *hyperperiod*, denoted by h. The total number of possible restarts in h can be calculated as $\lfloor \frac{h}{T_r} \rfloor$. Out of all the possible reboot instances, let us assume that $\tau_{r,k}$ is closest to $\tau_{i,m}$. Between $\tau_{i,m}$ and

$\tau_{r,k}$, there can be three possible relations: $kT_r \leq mT_i$, $mT_i < kT_r \leq (m+1)T_i$, and $kT_r > (m+1)T_i$.

Case 1: The worst-case overhead will occur at $kT_r = mT_i$, where the overhead will be $c_r = \epsilon + \epsilon'$.

Case 2: In this case, either $kT_r - mT_i$ is large enough for $\tau_{i,m}$ to complete the execution, or the reboot will terminate the task, and it will be deemed non-schedulable. Hence, the worst-case overhead will be 0 in this case.

Case 3: This case won't affect the execution of $\tau_{i,m}$. Hence, this case won't add an overhead to the WCRT.

Hence, the highest interference due to reboot will be observed in case 1. Therefore, taking $c_r = \epsilon + \epsilon'$ in Eq. V.2 captures the WCRT out of all possible cases.

□

The key here is that a task instance can face a maximum of one reboot during its execution. If the reboot preempts the task, the decision unit will switch the control to the safety unit to prevent the system from crashing and switch back to the complex controller when it is active after the reboot.

A task is deemed non-schedulable if any instance of the task fails to complete execution within the deadline. Lemma V.1 states the WCRT equation for a periodic reboot. The WCRT for a task can be used to formally state the conditions for schedulability.

Lemma V.2. *For a task τ_i to be schedulable in a secure reboot-enabled RTES, it is necessary to satisfy the following conditions:*

1. $R_i \leq T_i$,

2. $U + u_r \leq 1$, *where* u_r *is the utilization of* τ_r,

3. $R_i \leq T_r$,

If condition 1 is not satisfied, a task released at time mT_i will not be able to complete execution before the release of the next instance of the task at time $(m+1)T_i$. The following reasoning can explain condition 2: Let $U = \frac{c}{T}$, and $u_r = \frac{c_r}{T_r}$. If $\frac{c}{T} + \frac{c_r}{T_r} > 1$, it implies that $\frac{cT_r + c_rT}{TT_r} > 1$. We know that the hyperperiod of a task set is equal to the LCM of the period of all the tasks. Let us denote a hyperperiod by h. Hence, we can also write $\frac{cT_r + c_rT}{h} > 1$, or $cT_r + c_rT > h$, which implies that all the tasks (including the restart task) cannot be accommodated within the given hyperperiod if condition 2 is not satisfied.

Condition 3 of Lemma V.2 extends condition 1 and only applies to systems with periodic reboots. The conditions state that the reboot period has to be at least the length of the WCRT of the task, else the task will be terminated by the periodic reboot, and it will never be able to complete execution.

For a periodic task set, we can treat the necessary schedulability conditions stated in Lemma V.2 as the base condition for schedulability. However, these conditions are insufficient to prove a task's schedulability because they do not account for all the possible instances of a task. As stated in the proof of Lemma V.1, there can be cases where $kT_r - mT_i \leq R_i$ and the task will fail to complete before being terminated by the system reboot. We can generalize the cases from the proof of Lemma V.1 to define the *execution window* of the task.

Definition V.1 (Execution Window)**.** *For an arbitrary instance* m *of a task,* $\tau_{i,m}$*, the execution window is the maximum available execution time before the task gets terminated. The*

execution window of any arbitrary instance $\tau_{i,m}$ can be formally defined as:

$$X_{i,m} = \begin{cases} kT_r - mT_i & mT_i < kT_r \leq (m+1)T_r \\ T_i, & kT_r \leq mT_i \text{ or } kT_r > (m+1)T_i \end{cases} \tag{V.3}$$

For all possible pairwise values of (m, k), where m and k are instances of a system task and reboot task respectively.

In Fig. 5.2, we see three different cases of execution window. For $X_{i,1}$, the task executes after the system has rebooted into a fresh state, in this case, the execution window is T_i. In the case of $X_{i,2}$, we see no interference due to reboot and this case also has the same execution window size. However, in the case of $X_{i,3}$ we note that the execution time has been shortened due to the periodic reboot. Using the three cases, Eq. V.3 can be further tightened to only use instances of T_r to calculate the execution windows that are shortened due to the periodic reboot. We can define the minimum available execution window for an arbitrary task based on the reboot period.

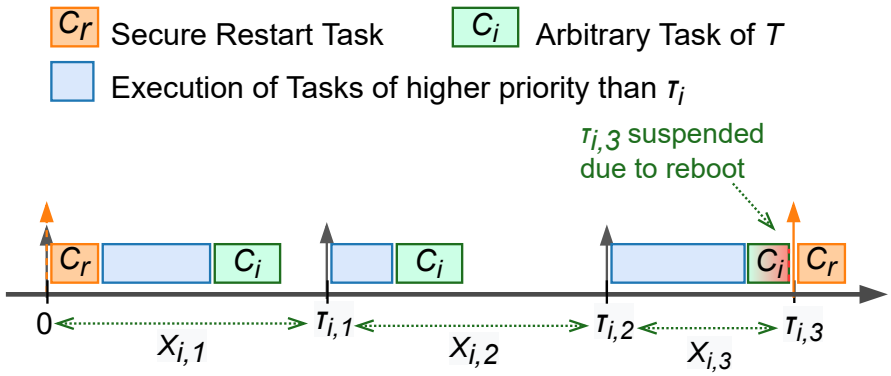


Figure 5.2: Example of three execution windows ($X_{i,1}$, $X_{i,2}$, and $X_{i,3}$) for task τ_i. The minimum execution window is $X_{i,3}$ because it is shorter than $X_{i,1}$ and $X_{i,2}$ due to the upcoming second instance of τ_r. [1]

Definition V.2 (Minimum Execution Window)**.** *The shortened execution window can be defined as:*

$$X_{i,m} = \begin{cases} kT_r - \left\lfloor \frac{kT_r}{T_i} \right\rfloor T_i, & kT_r \mod T_i \neq 0 \\ T_i, & kT_r \mod T_i = 0 \end{cases} \tag{V.4}$$

In the piecewise equation Eq. V.4, the conditions are based on the divisibility of kT_r by T_i. If the reboot instance is a multiple of T_i, i.e., $kT_r = nT_i$ for $n \in \mathbb{Z}$, the execution window of the immediately preceding task will be $kT_r - \frac{kT_r}{T_i}T_i = nT_i - nT_i = 0$, since $kT_r = nT_i$. Hence, the execution window available to the nth instance of the task is T_i as the reboot trigger coincides with the task deadline, and the reboot does not shorten the task window. Using Eq. V.4, the Minimum Execution Window can be defined as $min\{X_{i,m}\}_{m=0}^{N}$, where $N = \frac{h}{T_i}$.

Theorem V.1. *Suppose the WCRT of a periodic task τ_i, calculated using Lemma V.1, satisfies the conditions of Lemma V.2. In that case, the task is guaranteed to be schedulable if the minimum execution window is at least as long as the WCRT of the task.*

Proof. Let us assume that task τ_i satisfies Lemma V.2 where the WCRT, R_i, is calculated using Lemma V.1, which accounts for all the possible worst-case interference. Let us assume that the Minimum Execution Window $= X^{min}$. If $R_i < X^{min}$, then using Definition V.2, $R_i < X_{i,m} \forall 0 < m \leq \frac{h}{T_i}$, which says that if the WCRT of a task can be accommodated within the minimum execution window, the task can complete execution in all its instances. Recall that the schedulability of a task is defined as the ability to com-

plete execution before the deadline in all the instances of the task. Hence, Theorem V.1 sufficiently proves the schedulability of any task τ_i. □

5.4 Evaluation

In this section, we evaluate the impact of the addition of the secure reboot task on the task set schedulability. The evaluation setup and the related codes are open-sourced and published through GitHub [73]

5.4.1 Experimental Setup

We implemented our approach on the RTEMS RTOS [48] using *Das U-Boot*, a popular open-source bootloader compatible with a wide range of embedded devices. There are two steps to implementation; the first stage occurs on a host computer, and the second on the RTES. The U-boot bootloader is built during the first phase using user-provided configurations supplied through an `.its` file. Next, the hashed image is encrypted with RSA2048 and stored in a flattened image tree format. In addition to the flattened image tree, the device tree and generated public key are stored in a read-only memory location on the RTES. The second stage occurs every time the RTES is restarted. U-Boot uses the public key obtained from stage 1 to verify the hash of the kernel image and only allows a signature-verified image to boot. To ensure the integrity of execution, we terminate and discard all the tasks that did not complete execution before the reboot was triggered.

For performance analysis of the proposed model, we used Theorem V.1 on a synthetic task set that we generated using the UUnifast algorithm [74]. With a constant value for the

hyperperiod ($h = 1000$), we randomly selected task periods from the set of the factors of h. For each task τ_i, the WCET is calculated using $C_i = T_i \times U_i$. We varied U from 0.1 to 0.9 in steps of 0.1. For each value of U, we generated 1000 task sets with 20 tasks in each task set. The reboot overhead values, i.e., values for ϵ and ϵ', used in this performance analysis are recorded from our hardware test implementation discussed above.

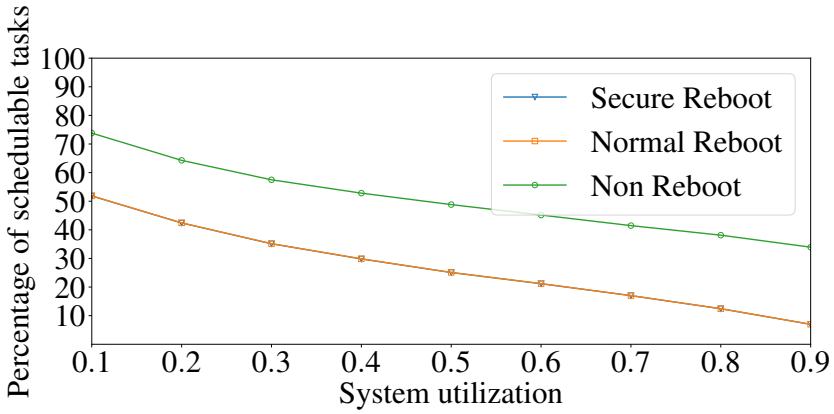
5.4.2 Experiments and Results

We performed the following experiments to gain quantitative insights about the performance overheads due to periodic secure reboots. We measured performance in terms of the impact on the system's schedulability in three different modes:

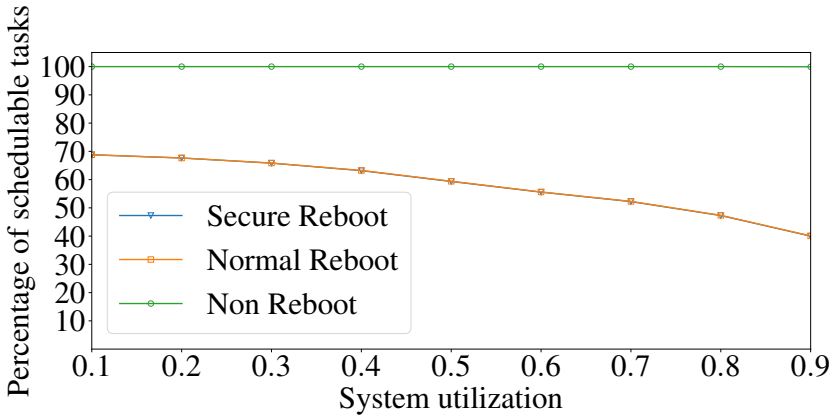
1. **No reboot:** $C_r = 0$
2. **Non-secure reboot:** $C_r = \epsilon$
3. **Secure-reboot:** $C_r = \epsilon + \epsilon'$

We define *schedulability* of a task set as the percentage of the tasks that can complete execution before their respective deadline. For each experiment, we used an arbitrarily fixed priority preemptive (AFPP) scheduling [70] and a rate monotonic (RM) scheduling algorithm [75] for assigning priorities to each task in the task sets.

Experiment 1: This experiment shows the schedulability comparison of a set of 1000 task sets for each value of U with a randomly chosen fixed value of T_r from a range of $(0, h]$. The reboot overheads are $\epsilon = 5.05$ and $\epsilon' = 0.02856$ seconds, which are collected from our implementation setup. Fig. 5.3 shows the impact of the periodic restart of the complex controller. Interestingly, we observe an almost indistinguishable pattern in the



(a) AFPP Scheduling



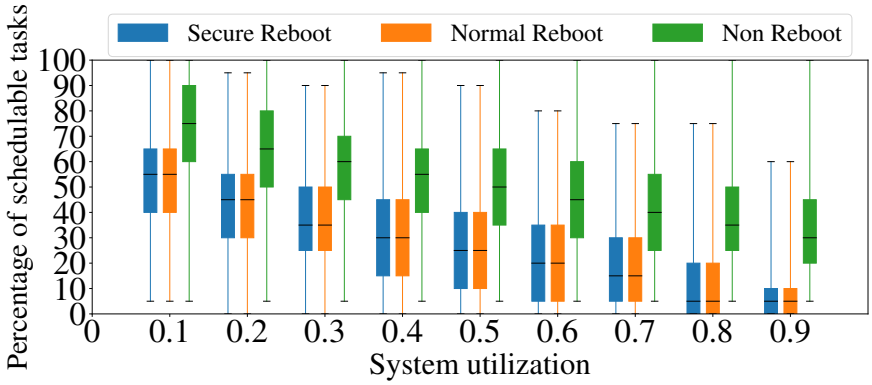
(b) RM Scheduling

Figure 5.3: Impact of Secure Boot on Task Set Schedulability using AFPP and RM Scheduling. [1]

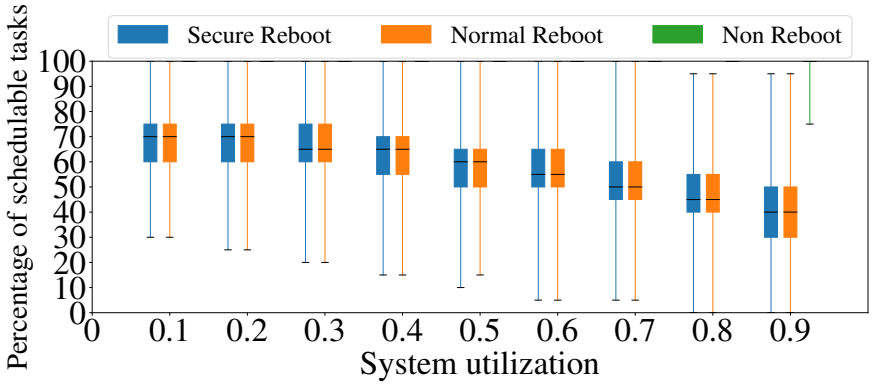
normal reboot and secure reboot traces with both AFPP and RM scheduling (see Figs. 5.3a and 5.3b), which implies that for restart-based systems, there is no significant reduction in schedulability by adding the secure boot sequence in the restart operation. In particular, for the utilization range typically used in CPS (0.5 to 0.7), the maximum drop in schedulability is approximately 0.03% for AFPP and 0.081% for RM scheduling. Hence, the secure boot can be feasibly added for real-time systems with an existing periodic restart-based mechanism without sacrificing schedulability.

Experiment 2: We extend Experiment 1 further and analyze the numerical summary from a randomly generated task set. To understand the performance trend better, we generated additional task sets and analyzed the schedulability of 50000 randomly generated tasks using boxplots. The five-point summary (minimum, first quartile, median, third quartile, and maximum) schedulability of all tasks shows a complete picture of the task schedulability at the given utilization level with the same fixed values of ϵ and ϵ' as used in Experiment 1. We used a pseudo-random number generator to assign values for T_r. Fig. 5.4 shows the schedulability of the task set with an arbitrary value of $T_r = 120$. The box plot demonstrates the relation between the system utilization and schedulability of task sets with a constant T_r.

Experiment 3: We now examine the impact of weighted schedulability [76] as a function of the utilization level and task schedulability at each utilization level. We define weighted schedulability as: $\frac{\sum_{i=0}^{k-1}(U_i.S(U_i,\epsilon+\epsilon',T_{r,m}))}{\sum_{i=0}^{k-1}U_i}$, where $S(U_i, \epsilon + \epsilon', T_{r,m})$ is the schedulability at utilization level U_i and $T_{r,m}$. The resulting plot in Fig. 5.5 shows that the schedulability of the task is directly proportional to the reboot period. On the one hand, a longer reboot period results in a higher frequency of the periodic reboot, which lowers the task schedulability. On the other hand, using a lower frequency of secure reboot increases the system's vulnerability. We notice a pattern of sudden peak (when the T_r is a factor of h) immediately followed by a steep drop in schedulability. This observation is because the task periods are factors of h. The same argument can explain the steep drop: when T_r takes values that are immediately after factors of h, every task having a period equal to a factor of h will have an instance released and terminated due to a reboot being triggered, which severely impacts the schedulability of the task sets for those values of T_r. From this exper-



(a) AFPP Scheduling

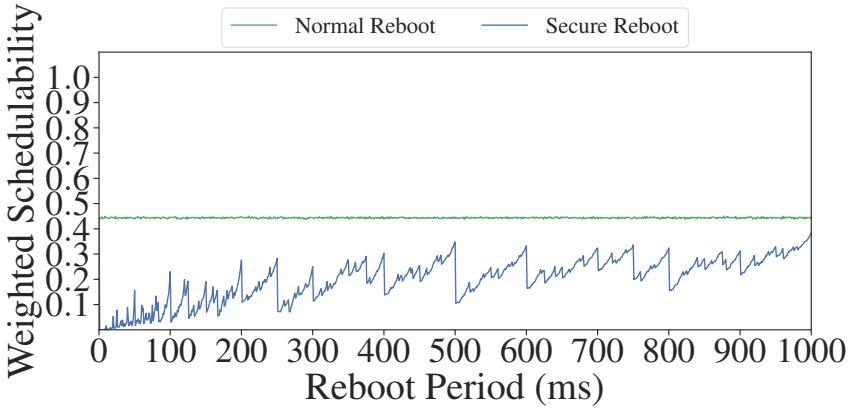


(b) RM Scheduling

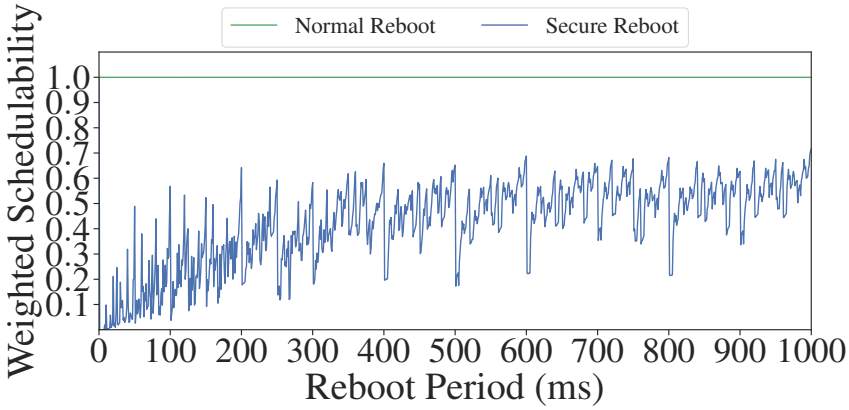
Figure 5.4: Impact of Secure Boot on Schedulability. [1]

iment and Experiment 1, we infer that the biggest impact on performance comes from the reboot period, and the schedulability can be maximized by setting the reboot period as the least common multiple of a subset of tasks. The subset of tasks can be selected based on factors such as priority, guaranteeing the execution of high-priority tasks. The task subset selection can also be done to execute the maximum number of tasks.

Experiment 4: In Experiments 1 and 2, we compare the schedulability with fixed reboot overhead and fixed reboot period. In Experiment 3, we observe the impact of the reboot period using a weighted schedulability plot with fixed reboot overhead. This exper-



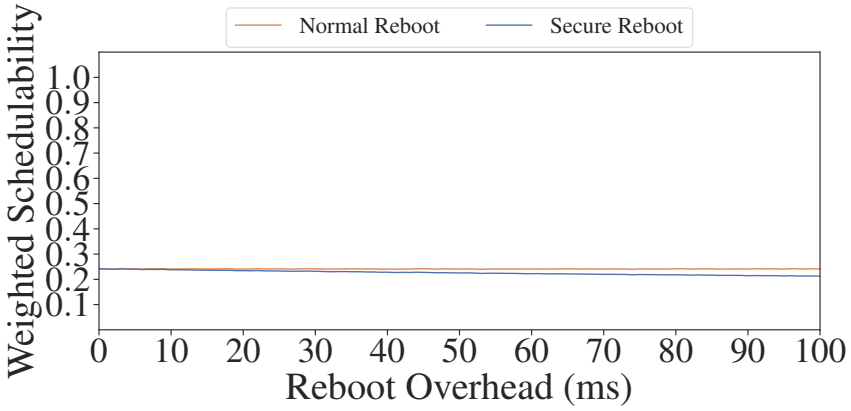
(a) AFPP Scheduling



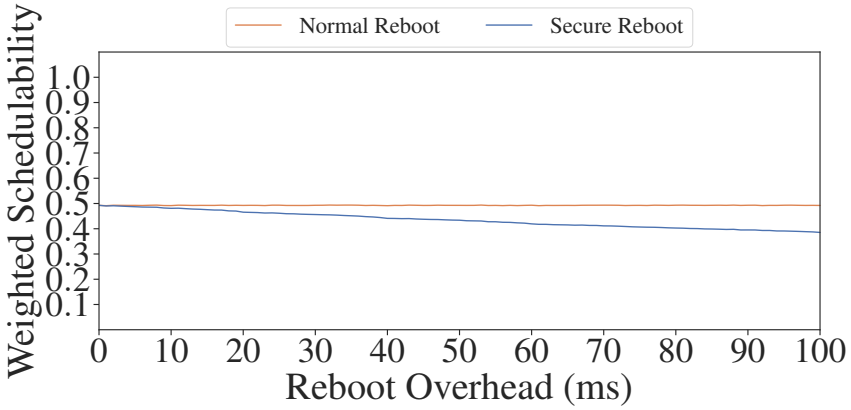
(b) RM Scheduling

Figure 5.5: Weighted schedulability as a function of reboot period T_r. [1]

iment demonstrates the impact of adding a secure reboot over a normal reboot. Fig. 5.6 shows a linear depreciation in the schedulability as the overhead percentage of the secure reboot increases. The plot generated from synthetic experiments shows a similar trend to Experiment 1, where we used values from a real system. In Fig. 5.6a, the schedulability difference with $\epsilon' = 0.01 \times \epsilon$ is around 0.03% which is close to what we found in Experiment 1. The weighted schedulability with RM scheduling in Fig. 5.6b also shows



(a) AFPP Scheduling



(b) RM Scheduling

Figure 5.6: Schedulability as a function of the secure reboot overhead ϵ'. [1]

a comparable value of 0.1% compared to 0.08% with real system values. Hence, based on the reboot overhead collected from hardware, the secure reboot-based mechanism has minimal tradeoff on system schedulability.

5.5 Summary

This chapter presents a secure boot mechanism for restart-based real-time CPS utilizing the Simplex architecture. Hardware-based evaluations offer a schedulability analysis of real-time task sets when secure boot is enabled. The experimental results demonstrate that periodic secure boot has a minimal impact when employing fixed-priority scheduling schemes under realistic reboot overhead conditions. This research establishes a foundation for incorporating widely recognized security features, such as secure boot, into the recovery processes of real-time systems. Future work may investigate the re-execution of terminated tasks, explore alternative scheduling paradigms that might align better with reboot scheduling, and consider the randomization of reboot timing.

CHAPTER VI

FUTURE WORK

This chapter presents potential future work to extend the research further and investigate the questions not addressed in this dissertation.

6.1 Improving Defense Against Side-Channels in RTES

This dissertation motivates the exploration of novel defense techniques capable of effectively safeguarding against the exploitation of standard application features for executing coordinated side-channel attacks. The adaptive execution of NOSYNEIGHBOR challenges several assumptions established in the leading attack literature by demonstrating that it is possible to acquire precise timing information of the target system without prior knowledge of additional system parameters.

Future research could also examine the effects of NOSYNEIGHBOR on various scheduler and task set models. Given that timing inference relies on the adaptability of the execution of informant tasks, different system workloads are expected to influence the outcomes of the inference in diverse ways. Additionally, the choice of scheduler may impact the in-

ference results, as certain schedulers may prove inherently more resilient to such adaptive side-channel attacks. A thorough examination of the performance-security tradeoff for different scheduler algorithms and workload composition can provide essential insights into improving the side-channel resilience of real-time systems.

6.2 Bypassing Trusted Execution Environment

A trusted execution environment (TEE) is a hardware-based isolation technique to mitigate side-channel inference. This dissertation assumes that the attack target does not have a TEE-based defense technique. However, many embedded targets are incorporating TEE to prevent side-channel and other types of attacks.

TEE provides an isolated memory space that only allows selected tasks to run in a secure environment. Future work can investigate extending NOSYNEIGHBOR to execute adaptive side-channel inference against a victim task running inside a secure environment in TEE.

6.3 Automated Selection of Software Components

The use of modularization in RTES, presented in Chapter IV, lays the ground for future work on modularizing other software components to enable rapid patching and independent security research in each of those components.

The dissertation further motivates exploration into developing metrics that can effectively quantify various software stacks' functional and non-functional attributes. Automation tools can use such metrics to facilitate the selection of the right set of software stacks

for each project. The metrics used in Chapter IV are based on the function requirements related to memory and the timing overhead. These metrics can be expanded further to calculate the overhead generated by subelements within each stack to relate the overhead evaluation with the specific type of workload that each project or target device might need.

CHAPTER VII

CONCLUSION

This dissertation introduces a novel side-channel attack technique called NOSY-NEIGHBOR which bypasses state-of-the-art defense mechanisms by leveraging the timing predictability of real-time systems. This attack enables the inference of timing parameters of tasks executing within separate partitions, effectively circumventing randomization-based defenses commonly used to mitigate side-channel vulnerabilities in real-time operating systems (RTOS). The demonstrated effectiveness of NOSYNEIGHBOR particularly in a testbed environment utilizing a Linux-based real-time operating system, underscores the potential risks adversaries can exploit.

Supply chain attacks continue to be a prominent method for propagating threats like NOSYNEIGHBOR with systems relying on legacy software particularly vulnerable due to longstanding flaws and vulnerabilities inherent to the legacy components. To address this challenge, the dissertation proposes a modularization technique known as networking-as-a-library. This approach facilitates transitioning into an upgraded network stack by allowing

the users to continue using legacy components through externally linked libraries. Hence, the system downtime is minimized during the transition.

Additionally, the research strengthens the security posture of RTEMS, which is used in multiple critical infrastructure sectors, including energy, defense, and transportation. The dissertation establishes theoretical bounds on the feasibility of integrating a proactive periodic recovery routine within real-time systems without compromising their real-time properties. This proposed recovery method incorporates a secure boot process at each system startup and ensures predictable periodic restarts, allowing the system to revert to a safe state with bounded overhead.

All research outcomes of this dissertation are open source and freely available for use and further research [4, 5, 73], promoting a collaborative effort to enhance security in real-time systems against evolving threats.

REFERENCES

[1] V. Banerjee, S. Hounsinou, H. Olufowobi, M. Hasan, and G. Bloom, "Secure reboots for real-time cyber-physical systems," in *Proceedings of the 4th Workshop on CPS & IoT Security and Privacy*, pp. 27–33, 2022.

[2] M. Hasan, A. Kashinath, C.-Y. Chen, and S. Mohan, "Sok: Security in real-time systems," *ACM Computing Surveys*, 2024.

[3] M.-K. Yoon, M. Liu, H. Chen, J.-E. Kim, and Z. Shao, "Blinder: Partition-Oblivious hierarchical scheduling," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 2417–2434, USENIX Association, Aug. 2021.

[4] V. Banerjee, "Rtems net-legacy." https://gitlab.rtems.org/opticron/rtems-net-legacy.

[5] V. Banerjee, "Rtems net-lwip." https://gitlab.rtems.org/rtems/pkg/rtems-lwip.

[6] C.-Y. Chen, A. Ghassami, S. Nagy, M.-K. Yoon, S. Mohan, N. Kiyavash, R. B. Bobba, and R. Pellizzoni, "Schedule-based side-channel attack in fixed-priority real-time systems," 2015.

[7] C.-Y. Chen, S. Mohan, R. Pellizzoni, R. B. Bobba, and N. Kiyavash, "A Novel Side-Channel in Real-Time Schedulers," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 90–102, 2019.

[8] S. Hounsinou, M. Stidd, U. Ezeobi, H. Olufowobi, M. Nasri, and G. Bloom, "Vulnerability of controller area network to schedule-based attacks," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, pp. 495–507, IEEE, 2021.

[9] S. Liu and W. Yi, "Task parameters analysis in schedule-based timing side-channel attack," *IEEE Access*, vol. 8, pp. 157103–157115, 2020.

[10] Ş. Vădineanu and M. Nasri, "Robust and accurate period inference using regression-based techniques," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, pp. 358–370, IEEE, 2020.

[11] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha, "TaskShuffler: A Schedule Randomization Protocol for Obfuscation against Timing Inference Attacks in Real-Time Systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 1–12, 2016.

[12] M.-K. Yoon, J.-E. Kim, R. Bradford, and Z. Shao, "Taskshuffler++: Real-time schedule randomization for reducing worst-case vulnerability to timing inference attacks," *arXiv preprint arXiv:1911.07726*, 2019.

[13] C.-Y. Chen, M. Hasan, A. Ghassami, S. Mohan, and N. Kiyavash, "Reorder: Securing dynamic-priority real-time systems using schedule obfuscation," *arXiv preprint arXiv:1806.01393*, 2018.

[14] K. Krüger, G. Fohler, and M. Volp, "Improving security for time-triggered real-time systems against timing inference based attacks by schedule obfuscation," in *Work-in-Progress Proceedings ECRTS'17*, 2017.

[15] 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)*, pp. 103–116, IEEE, 2019.

[16] M.-K. Yoon, J.-E. Kim, R. Bradford, and Z. Shao, "TimeDice: Schedulability-Preserving Priority Inversion for Mitigating Covert Timing Channels Between Real-time Partitions," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 453–465, June 2022. ISSN: 2158-3927.

[17] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The system-level simplex architecture for improved real-time embedded system safety," in *2009 15th IEEE RTAS*, pp. 99–107, IEEE, 2009.

[18] F. A. T. Abad, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo, "Reset-based recovery for real-time cyber-physical systems with temporal safety constraints," in *2016 IEEE 21st ETFA*, pp. 1–8, IEEE, 2016.

[19] P. Jagtap, F. Abdi, M. Rungger, M. Zamani, and M. Caccamo, "Software fault tolerance for cyber-physical systems via full system restart," *ACM Transactions on Cyber-Physical Systems*, vol. 4, no. 4, pp. 1–20, 2020.

[20] F. Abdi, M. Hasan, S. Mohan, D. Agarwal, and M. Caccamo, "Resecure: A restart-based security protocol for tightly actuated hard real-time systems," *IEEE CERTS*, pp. 47–54, 2016.

[21] 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 *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 10–21, IEEE, 2018.

[22] F. A. T. Abad, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo, "Reset-based recovery for real-time cyber-physical systems with temporal safety constraints," in *2016 IEEE 21st ETFA*, pp. 1–8, 2016.

[23] F. Abdi, C.-Y. Chen, M. Hasan, S. Liu, S. Mohan, and M. Caccamo, "Restart-based security mechanisms for safety-critical embedded systems," *arXiv preprint arXiv:1705.01520*, 2017.

[24] R. Romagnoli, B. H. Krogh, and B. Sinopoli, "Design of software rejuvenation for cps security using invariant sets," in *2019 American Control Conference (ACC)*, pp. 3740–3745, IEEE, 2019.

[25] L. Sha *et al.*, "Using simplicity to control complexity," *IEEE Software*, vol. 18, no. 4, pp. 20–28, 2001.

[26] S. Bak, K. Manamcheri, S. Mitra, and M. Caccamo, "Sandboxing controllers for cyber-physical systems," in *2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, pp. 3–12, IEEE, 2011.

[27] S. Bak, T. T. Johnson, M. Caccamo, and L. Sha, "Real-time reachability for verified simplex design," in *2014 IEEE RTSS*, pp. 138–148, IEEE, 2014.

[28] F. Abdi, R. Tabish, M. Rungger, M. Zamani, and M. Caccamo, "Application and system-level software fault tolerance through full system restarts," in *2017 ACM/IEEE 8th ICCPS*, pp. 197–206, IEEE, 2017.

[29] M. Arroyo, H. Kobayashi, S. Sethumadhavan, and J. Yang, "Fired: frequent inertial resets with diversification for emerging commodity cyber-physical systems," *arXiv preprint arXiv:1702.06595*, 2017.

[30] M. A. Arroyo, M. T. I. Ziad, H. Kobayashi, J. Yang, and S. Sethumadhavan, "Yolo: frequently resetting cyber-physical systems for security," in *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, vol. 11009, p. 110090P, International Society for Optics and Photonics, 2019.

[31] M. Weiß, B. Heinz, and F. Stumpf, "A cache timing attack on aes in virtualization environments," in *Financial Cryptography and Data Security: 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers 16*, pp. 314–328, Springer, 2012.

[32] K. Krüger, M. Volp, and G. Fohler, "Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems," *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 106, p. 22, 2018.

[33] B. Hammi, S. Zeadally, and J. Nebhen, "Security threats, countermeasures, and challenges of digital supply chains," *ACM Computing Surveys*, 2023.

[34] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pp. 23–43, Springer, 2020.

[35] J. Chen, T. Kloda, A. Bansal, R. Tabish, C.-Y. Chen, B. Liu, S. Mohan, M. Caccamo, and L. Sha, "SchedGuard: Protecting against Schedule Leaks Using Linux Containers," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 14–26, 2021.

[36] J. Chen, T. Kloda, R. Tabish, A. Bansal, C.-Y. Chen, B. Liu, S. Mohan, M. Caccamo, and L. Sha, "Schedguard++: Protecting against schedule leaks using linux containers on multi-core processors," *ACM Trans. Cyber-Phys. Syst.*, vol. 7, feb 2023.

[37] B. B. Brandenburg, *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[38] R. West, Y. Li, E. Missimer, and M. Danish, "A virtualized separation kernel for mixed-criticality systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 3, pp. 1–41, 2016.

[39] A. Danko, "Adaptive partitioning scheduler for multiprocessing system," Jan. 14 2014. US Patent 8,631,409.

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

[41] H. Edelsbrunner and H. A. Maurer, "On the intersection of orthogonal objects," *Information Processing Letters*, vol. 13, no. 4-5, pp. 177–181, 1981.

[42] E. M. McCreight, "Efficient algorithms for enumerating intersecting intervals and rectangles," tech. rep., Xerox Palo Alto, 1980.

[43] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005. Publisher: Springer.

[44] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 20, pp. 46–61, Jan. 1973.

[45] M. Gordon and M. Kochen, "Recall-precision trade-off: A derivation," *Journal of the American Society for Information Science*, vol. 40, no. 3, pp. 145–151, 1989.

[46] M. Buckland and F. Gey, "The relationship between recall and precision," *Journal of the American society for information science*, vol. 45, no. 1, pp. 12–19, 1994.

[47] V. Banerjee, S. Hounsinou, H. Gerber, and G. Bloom, "Modular network stacks in the real-time executive for multiprocessor systems," in *2021 Resilience Week (RWS)*, pp. 1–7, IEEE, 2021.

[48] G. Bloom, J. Sherrill, T. Hu, and I. C. Bertolotti, *Real-Time Systems Development with RTEMS and Multicore Processors*. CRC Press, Nov. 2020.

[49] G. Bloom and J. Sherrill, "Scheduling and Thread Management with RTEMS," *SIGBED Rev.*, vol. 11, pp. 20–25, Feb. 2014.

[50] C. Vinschen and J. Johnston, "The newlib homepage," 2018.

[51] W. Gatliff, "Porting and using newlib in embedded systems."

[52] C. Johns, J. Sherrill, B. Gras, S. Huber, and G. Bloom, "FreeBSD and RTEMS, UNIX in a Real-time Operating System," *FreeBSD Journal*, 2016.

[53] B. Beranek, "A history of the arpanet: the first decade," *Technical report*, 1983.

[54] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[55] "Freebsd security."

[56] A. Dunkels, "Design and implementation of the lwip tcp/ip stack," *Swedish Institute of Computer Science*, vol. 2, no. 77, 2001.

[57] M. Hamad and V. Prevelakis, "Implementation and performance evaluation of embedded ipsec in microkernel os," in *2015 World Symposium on Computer Networks and Information Security (WSCNIS)*, pp. 1–7, IEEE, 2015.

[58] L. Mejdrech, "Networking and tcp/ip stack for helenos system," *Univerzita Karlova, Matematicko-fyzikální fakulta*, 2010.

[59] A. Kantee, "The design and implementation of the anykernel and rump kernels," *Aalto university*, 2016.

[60] "Porting lwIP - FreeRTOS."

[61] G. Bloom and J. Sherrill, "Harmonizing ARINC 653 and Realtime POSIX for Conformance to the FACE Technical Standard," in *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 98–105, May 2020. ISSN: 2375-5261.

[62] N. Schild and C. Scheuer, "Embedded ipsec, light weight ipsec implementation," *Diplome Thesis, Berne Univ, Switzerland*, 2003.

[63] T. Nagy, "The Waf Book."

[64] FreeBSD.org, "Chapter 34. Advanced Networking."

[65] uLan, "ulan protocol for rs-485 9-bit network / lwip-omk / [9e6ce8]."

[66] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (USA), p. 41, USENIX Association, 2005.

[67] T. Straumann, "qemu + uc5282," 2009.

[68] L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving dependable real-time systems," in *1996 IEEE AeroConf.*, vol. 1, pp. 335–346, IEEE, 1996.

[69] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The system-level simplex architecture for improved real-time embedded system safety," in *2009 15th IEEE RTAS*, pp. 99–107, IEEE, 2009.

[70] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *[1990] Proceedings 11th Real-Time Systems Symposium*, pp. 201–209, IEEE, 1990.

[71] M. Gonzalez, H. Mark, H. Klein, and J. P. Lehoczky, "Fixed priority scheduling of periodic tasks with varying execution priority," in *In Proceedings, IEEE Real-Time Systems Symposium*, Citeseer, 1991.

[72] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software engineering journal*, vol. 8, no. 5, pp. 284–292, 1993.

[73] V. Banerjee, "Rtems secboot." https://github.com/thelunatic/secboot.

[74] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.

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

[76] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," *Proceedings of OSPERT*, vol. 10, pp. 33–44, 2010.