

© 2020 Monowar Hasan

INTEGRATING SECURITY INTO REAL-TIME CYBER-PHYSICAL SYSTEMS

BY

MONOWAR HASAN

DISSERTATION

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

Urbana, Illinois

Doctoral Committee:

Professor Sibin Mohan, Chair
Professor Klara Nahrstedt
Professor Tarek Abdelzaher
Professor Kirill Levchenko
Professor Rodolfo Pellizzoni
Professor Rakesh Bobba

ABSTRACT

Cyber-physical systems (CPS) such as automobiles, power plants, avionics systems, unmanned vehicles, medical devices, manufacturing and home automation systems have distinct *cyber* and *physical* components that must work cohesively with each other to ensure correct operation. Many cyber-physical applications have “real-time” constraints, i.e., they must function correctly within predetermined time scales. A failure to protect these systems could result in significant harm to humans, the system or even the environment. While traditionally such systems were isolated from external accesses and used proprietary components and protocols, modern CPS use off-the-shelf components and are increasingly interconnected, often via networks such as the Internet. As a result, they are exposed to additional attack surfaces and have become increasingly vulnerable to cyber attacks. Enhancing security for real-time CPS, however, is not an easy task due to limited resource availability (e.g., processing power, memory, storage, energy) and stringent timing/safety requirements. Security monitoring techniques for cyber-physical platforms (*a*) must execute with existing real-time tasks, (*b*) operate without impacting the timing and safety constraints of the control logic and (*c*) have to be designed and executed in a way that an adversary cannot easily evade it. The objective of my research is to *increase security posture of embedded real-time CPS by integrating monitoring/detection techniques that defeat cyber attacks without violating timing/safety constraints* of existing tasks. My dissertation work explores the real-time security domain and shows that *by employing a combination of multiple scheduling/analysis techniques and interactions between hardware/software-based security extensions, it becomes feasible to integrate security monitoring mechanisms in real-time CPS without compromising timing/safety requirements of existing tasks*. In this research, I (*a*) develop techniques to *raise the responsiveness of security monitoring tasks by increasing their frequency of execution*, (*b*) design a *hardware-supported framework to prevent falsification of actuation commands* — i.e., commands that control the state of the physical system and (*c*) propose *metrics to trade-off security with real-time guarantees*. The solutions presented in this dissertation *require minimal changes* to system components/parameters and thus *compatible for legacy systems*. My proposed frameworks and results are evaluated through both, *simulations* and *experiments on real off-the-shelf cyber-physical platforms*. The development of analysis techniques and design frameworks proposed in this dissertation will inherently make such systems more *secure* and hence, *safer*. I believe my dissertation work will bring researchers and system engineers one step closer to understand how to integrate two seemingly diverse yet important fields — real-time CPS and cyber-security — while gaining a better understanding of both areas.

To my amazing parents and in-laws, amicable sister, fabulous uncle, awesome grandmother (nana) and gorgeous wife — whose affections, supports, encouragements and prays make me successful.

ACKNOWLEDGMENTS

This dissertation is the culmination of many years of hard work and I could not achieve this without the companionship, inspiration and support of an entire community. I am grateful to have the opportunity to work with and learn from outstanding individuals. While I try to acknowledge every single one of them, I may miss a few names who helped me along the way.

First and foremost, I would like to thank my advisor Professor Sibin Mohan for trusting me and providing necessary mentoring as well as intellectual support. No single page of this dissertation would have been written without his support. I learned a lot from Prof. Mohan — from conducting research and developing presentation skills to mentoring and managing a large research group and whatnot. I am particularly grateful for valuing my opinions and supporting me during difficult times. I believe these lessons will be valuable in my future endeavor.

I would like to thank my Ph.D. committee members Professors Klara Nahrstedt, Tarek Abdelzaher, Kirill Levchenko, Rodolfo Pellizzoni and Rakesh Bobba who were more than generous with their precious time and feedback that significantly helped to improve the quality of this work. A special thanks to my collaborators Prof. Pellizzoni and Prof. Bobba — our countless hours of brainstorming and discussion sessions shape this work. I am also deeply indebted for their help in my academic job search process and writing reference letters for me. I am beholden to my master's advisor Professor Ekram Hossain for his supports and guidance in different stages of my academic career. I would not have ended up getting a Ph.D. from University of Illinois at Urbana-Champaign (UIUC) without his aspirations. Prof. Hossain also guided and actively assisted me throughout the complex job search process.

I would also like to thank my internship mentors — Dr. Ulf Lindqvist and Dr. Gabriela Ciocarlie from SRI International as well as Dr. Takayuki Shimizu and Dr. Hongsheng Lu from Toyota Motor North America — for giving me the opportunity to work on some really exciting research ideas.

Graduate school can never be pleasant experience without the support from administration. I would like to acknowledge and thank UIUC Computer Science department and Coordinated Science Laboratory (CSL) for providing me necessary assistance to conduct my research. Special thanks go to graduate academic advisors Viveka Kudaligama, Kara MacGregor and Maggie Chappell as well as CSL staff members Tonia Siuts, Kelli Anderson and Theron Seckington. Dr. Derek Attig and Mike Firmand from UIUC Graduate College examined and provided valuable feedback on my job application materials — thank you, it helped a lot! I also acknowledge the supports from funding agencies and our industry collaborators — National Science Foundation, Office of Naval Research, Cyber Resilient Energy Delivery Consortium and Toyota Motor North America — for the funding that have made my research possible.

I am fortunate to have the company of my colleagues, Dr. Chien-Ying Chen, Dr. Fardin Abdi, Ashish Kashinath, Kyo Hyun Kim and other SyNeRCyS@Illinois group members — the

collaborative work we did and the time we spent together had been a blissful experience. I hope we will keep in touch and continue to collaborate in the future.

I would not be here today, doing what I am, without my family. I will be always grateful to my parents Nazrul Islam and Dr. Monowara Begum, my in-laws Dr. Humayun Kabir and Noorjahan Morshead, my awesome grandmother (nana) Anowara Begum, my uncle Rabiul Alam and my sister Dr. Fahim Newsheen who have been my biggest support and who offered all their love and compassion — thank you for being there for me. I want to specially thank my wonderful wife, Syeda Noor E Sumaiya. I feel incredibly lucky to have such a supportive and loving partner. I could not have asked for a better partner — she is the most supportive person at times of stress and uncertainty. I would have been a drop-out in first year of my Ph.D. without her. It is my pleasure to endure this long journey with her — tasting the sweets and bitters of life together.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
1.1	Thesis Statement	4
1.2	Summary of Solutions	5
CHAPTER 2	RELATED WORK	9
2.1	Scheduling and Period Optimization in RTS	9
2.2	Security in Real-Time Cyber-Physical Systems	10
CHAPTER 3	ASSUMPTIONS ON ADVERSARIAL CAPABILITIES	12
3.1	Adversary Model for Chapter 4 and Chapter 5	12
3.2	Adversary Model for Chapter 6	12
CHAPTER 4	AN ADAPTIVE FRAMEWORK FOR INTEGRATING SECURITY TASKS IN SINGLE CORE REAL-TIME SYSTEMS	14
4.1	Introduction	14
4.2	Overview of CONTEGO	17
4.3	System Model	18
4.4	Period Adaptation	20
4.5	The Security Server	22
4.6	Algorithm	27
4.7	Evaluation	27
4.8	Conclusion	35
CHAPTER 5	A DESIGN-SPACE EXPLORATION FOR INTEGRATING SECURITY TASKS IN MULTICORE REAL-TIME SYSTEMS	36
5.1	Introduction	36
5.2	Model and Assumptions	38
5.3	HYDRA: Fixed Assignment of Security Tasks	39
5.4	HYDRA-C: Continuous Security Monitoring	42
5.5	Evaluation	47
5.6	Conclusion	52
CHAPTER 6	SELECTIVE CHECKING AND TRUSTED EXECUTION TO PREVENT FALSE ACTUATIONS IN REAL-TIME CYBER-PHYSICAL SYSTEMS	53
6.1	Introduction	53
6.2	Motivation, Overview and Background	55
6.3	Checking Actuation Commands	60
6.4	Game-Theoretic Analysis for Random Checking	62
6.5	Evaluation	68
6.6	Conclusion	79
CHAPTER 7	DISCUSSION	80

CHAPTER 8	CONCLUSION AND FUTURE WORK	82
8.1	Future Directions	84
APPENDIX A	CONTEGO – SUPPLEMENTARY MATERIALS	86
A.1	Linear Lower-Bound Supply Function and Schedulability Constraints	86
A.2	Solution to the Optimization Problems	88
A.3	Comparison with Exact Method	91
APPENDIX B	HYDRA/HYDRA-C – SUPPLEMENTARY MATERIALS	94
B.1	Solution to the Period Selection Problem in HYDRA	94
B.2	Comparing HYDRA with Optimal Multicore Assignment	94
B.3	Proof of Lemma 5.1	95
APPENDIX C	SCATE – SUPPLEMENTARY MATERIALS	96
C.1	System Model: Real-Time Task and Scheduling	96
C.2	Feasibility Conditions	97
C.3	Design-Time Tests for Integrating Actuation Checking in Existing Systems	97
C.4	Impact of Physical Inertia	98
REFERENCES	100

CHAPTER 1: INTRODUCTION

Cyber-physical systems (CPS) such as avionics, nuclear power plants, automobiles, space vehicles, power generation and distribution systems, medical devices, industrial robots have been in existence for decades. Such systems, which often have *safety-critical* properties and *real-time* (i.e., stringent timing) requirements. Any problems in real-time systems (RTS) could result in significant harm to humans, the system or even the environment. Systems with real-time requirements need to function correctly, but *within their predefined timing constraints*, often termed as “deadlines”. For example, consider real-time CPS applications with stringent timing constraints such as deployment of a car’s airbag or an industrial robot operates on a manufacturing conveyor line. A typical window for an airbag deployment (time between detection of collision and final airbag operations) is around 50–60 ms [1] and the response time for robot movements (i.e., placing and moving objects on the conveyor) is around 50–100 ms [2].

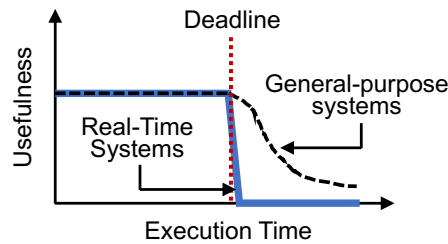


Figure 1.1: Timeliness requirements of real-time systems.

If real-time applications failed to comply with their timing requirements (deadlines), the usefulness of results produced by the system drops sharply (see Fig. 1.1). From the earlier airbag deployment and manufacturing robot example, if the application tasks fail to deploy the airbag in time or there is a delay to update the angle of rotation of the robot arm, the physical system may not work properly and hence put the safety of the human operators at risk. This is different from general-purpose systems where the usefulness drops in a more gradual manner (e.g., a web service may tolerate a few millisecond delay without degrading user experience significantly). Some of the common properties and assumptions related to RTS are as follows: *(i)* stringent timing and safety requirements, *(ii)* implemented as a system of periodic tasks, *(iii)* worst-case bounds are known for most loops as well as the critical pieces of code, *(iv)* no dynamically loaded or self-modifying code, *(v)* recursion is either not used or statically bounded and *(vi)* memory and processing power often limited.

Figure 1.2 presents a high-level illustration of a real-time CPS. Each real-time application in the system (called “task”) represents a time-critical function and a collection of such tasks are hosted on a hardware platform. The scheduler in real-time operating system (RTOS) uses timers and interrupt handlers to enforce timing guarantees at runtime. This ability of the scheduler to interrupt application processing at precise time instants is essential to ensure the “correctness” of

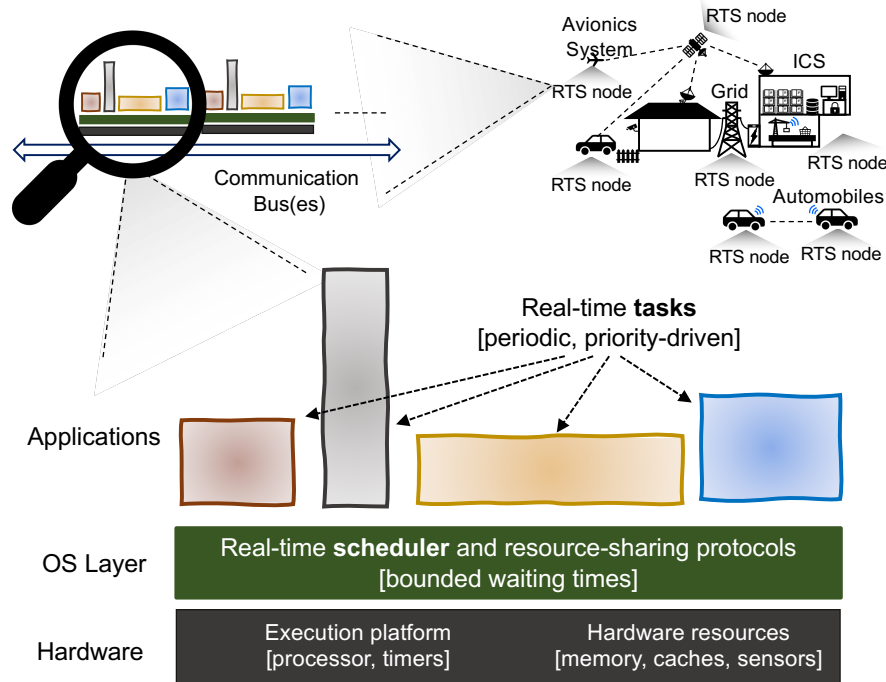


Figure 1.2: Abstraction of a real-time CPS node with common uses cases.

the system. Access to shared platform resources (such as caches, buses, memory) is regulated using resource sharing protocols to ensure data consistency and bounds on waiting time so that deadlines can be met. The communication network in RTS is required to provide service with low jitters and meet end-to-end message deadline for all messages.

Although safety and fault-tolerance have long been important design focus in such systems, security has rarely been a consideration in the design of real-time CPS mainly due to beliefs such as: (a) RTS lack inherent value to adversaries (“*why would anyone attack them?*”), (b) the prevalence of custom hardware/software/protocols will deter attackers (“*these protocols/hardware/software are secret and so arcane that no one can decipher them*”) and also (c) the lack of computing power and memory in these systems will throttle potential adversarial actions (“*what can they do even if they get in?*”). While traditionally RTS adopted proprietary protocols, platforms, software and were air-gapped (i.e., not connected to the outside world), with the advent of newer domains such as autonomous vehicles, drones, remote monitoring and control and Internet-of-Things (IoT)-specific applications, RTS find themselves front and center in modern society. Since many RTS now use commodity-off-the-shelf (COTS) components and are often connected to each other or even the Internet, they expose additional attack surfaces, often overturning all of the aforementioned beliefs. A number of high-profile attacks on real systems (e.g., Stuxnet [3], BlackEnergy [4]), attack demonstrations by researchers on automobiles [5, 6] and medical devices [7] have shown that the threat is real and systems composed of RTS might be vulnerable to cyber attacks.

Given the time and resource constraints under which RTS operate, vulnerabilities in RTS differ

considerably from those of traditional enterprise systems. Threats faced by RTS could vary in scope and effect; from the leakage of critical data [8] to hostile actions due to lack of authentication [5, 6, 9]. However, simply adding security mechanisms that provide confidentiality (e.g., encryption), integrity protection (e.g., message authentication) and availability (e.g., replication) without considering the real-time and embedded nature of such systems will not be effective. In the last few years there has been a lot of focus on securing critical CPS [10–18]. A major focus of such work has been on securing communication protocols and on monitoring and detection mechanisms at the network and application level. Given the increasing cyber-attack risks, however, it is essential to have a layered defense and integrate resilience against cyber attacks into the design of controllers and actuators (i.e., embedded RTS). It is also critical to retrofit existing controllers and actuators with protection, detection, survival and recovery mechanisms. In addition, stringent timing constraints severely inhibit how security solutions can be added to RTS; for instance, the protection methods should not cause timing problems in RTS.

Integrating security techniques in real-time CPS, however, is not an easy task since they have stringent timing and safety constraints — hence a security mechanism must not violate these requirements. Any security mechanisms have to co-exist with the real-time tasks in the system. However, stringent timing constraints in RTS introduce complexities — the strict deadlines for the real-time tasks may not allow for frequent execution of security mechanisms. Unlike in conventional IT settings, it may not be possible to execute the detection/monitoring tasks for arbitrary lengths of time. Designers of the systems are required to balance between security requirements (e.g., having enough cycles for effective monitoring detection) and the timing/safety requirements (i.e., not interfere with real-time deadlines). Further, it may not be feasible for legacy systems to adjust the parameters (such as run-times, period and execution order) of real-time tasks to accommodate security techniques. For example, how often and how long should a monitoring and detection task run to be effective but not interfere with real-time control or other safety-critical tasks? Integrating security into multicore real-time platforms is more challenging when compared to the single core systems since designers have multiple choices across cores to retrofit security mechanisms. While this real-time vs. security trade-offs could potentially be addressed for newer systems at design time, this is especially challenging for retrofitting legacy systems where the real-time tasks are already in place and perhaps cannot be modified. An understanding of the interplay between real-time constraints and the security requirements is very important for properly integrating the two fields. The focus of my research is to develop techniques that will allow us to trade back and forth between these, seemingly, conflicting properties so that designers of the systems can correctly gauge system requirements — hence, perhaps, meeting both, the real-time constraints as well as integrating effective security mechanisms.

My dissertation work presents *design-time frameworks for integrating security monitoring mechanisms into real-time CPS without violating timing/safety constraints*. In particular, I *study and develop models for integrating monitoring and detection mechanisms into legacy (i.e., existing) real-time CPS built using both, single and multicore processors*. The security mechanisms to

be integrated could be any detection/protection/recovery mechanism depending on the system requirements — for instance, an intrusion detection task or a trusted module that checks specific system signals (such as filesystems, network packets, actuation commands). As part of this research I analyze the various parameters that affect design choices while integrating security into RTS. I further develop *metrics* that will enable us to measure success and objectively analyze different solutions.¹ I also evaluate my solutions on a variety of platforms — from simulation engines to real hardware, viz., a ground rover, a flight controller, a infusion pump and a robotic arm.

1.1 THESIS STATEMENT

As mentioned earlier, integrating security in real-time CPS is not straightforward since monitoring/detection mechanisms must (a) co-execute with existing real-time tasks, (b) comply with timing/safety constraints and (c) designed/scheduled in a way that an adversary cannot easily evade them. The real-time security solutions are constrained to address the following challenges:

1. How do we integrate and then characterize the effects of security in real-time CPS those designed using both, single and multicore chips?
2. What are the trade-offs on the security and timing requirements while guaranteeing no (or minimal) perturbations for the real-time properties?
3. What are the performance criteria and metrics that need to be considered while integrating security into real-time CPS?

The challenges to answering these questions reside in the timing constraints imposed on real-time CPS. I address these challenges by postulating the following hypothesis:

It is possible to integrate security into real-time cyber-physical systems by a careful (task/scheduler-level) analysis of, and co-design with, system constraints, viz. software, hardware and timing requirements.

My dissertation work shows that it becomes feasible to integrate security mechanisms in real-time CPS without compromising timing/safety requirements by employing a combination of time-aware solutions such as, (i) by imposing scheduling-level constraints (Chapters 4–5) or (ii) interacting between scheduling/optimization techniques and hardware/software-based security extensions (Chapter 6). The end goal, then, is to provide designers with *a knob that they can use to tune to one side or the other — real-time vs. security.*

¹The development of metrics for security is hard in general but can be made in specialized domains, as I demonstrate.

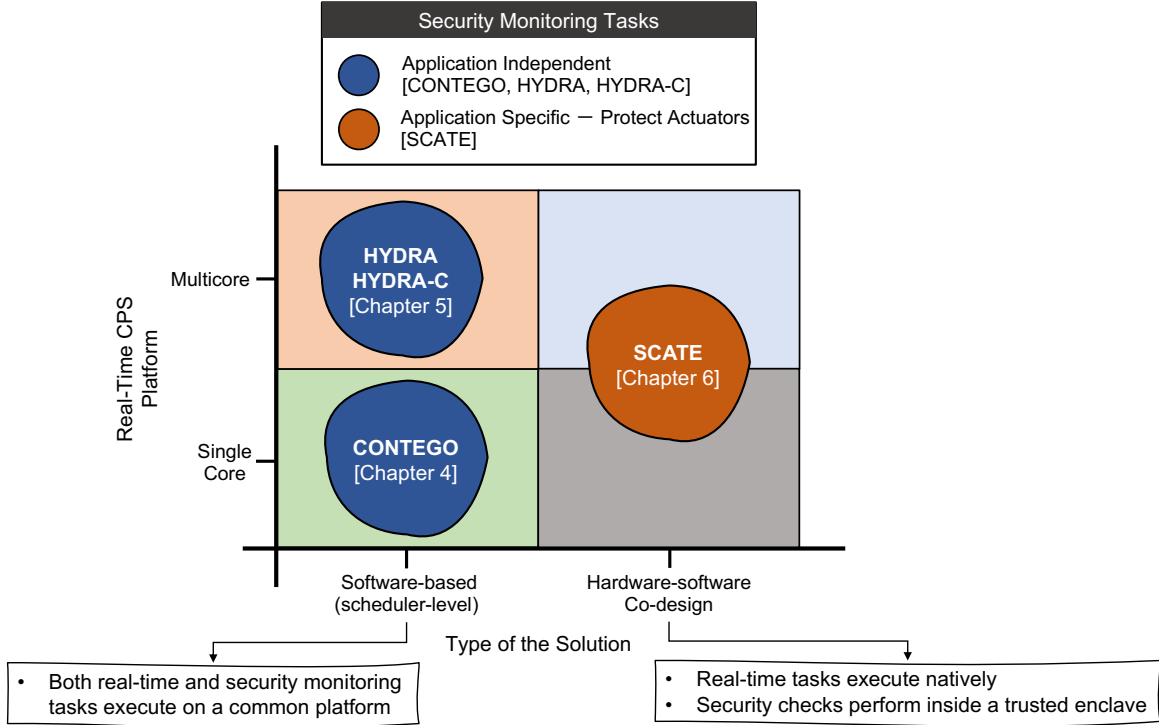


Figure 1.3: Various security integration techniques proposed in this dissertation. Chapters 4–5 present pure software-based solutions. CONTEGO (Chapter 4) is designed for single core systems while HYDRA and HYDRA-C (Chapter 5) target multicore platforms. Chapter 6 (SCATE) presents a hardware-supported architecture and compatible for both, single and multicore systems.

1.2 SUMMARY OF SOLUTIONS

Based on the aforementioned challenges, my dissertation work is divided into three parts that are presented in Chapters 4–6 (see Fig. 1.3). At the high-level, the solutions proposed in this dissertation can be classified along two major dimensions. The horizontal dimension in Fig. 1.3 represents the type of the security integration techniques and the vertical dimension represents the underlying real-time computational platform. The security integration techniques can be either (i) pure software-based (Chapters 4–5) or (ii) leverage hardware-assisted security features (Chapter 6). My proposed solutions are either (i) core-specific, i.e., designed for single core (Chapter 4) or multicore (Chapter 5) systems or (ii) applicable for both, single and multicore platforms (Chapter 6). I further consider cases where security monitoring tasks can be either (i) periodic and independent of real-time applications (Chapters 4–5) or (ii) perform application-specific checks, i.e., protect physical actuators from false actuation commands (Chapter 6). I now summarize the solutions proposed in this dissertation.

Chapters 4–5 present scheduler-level defense mechanisms. These techniques (a) are software-based approaches (integrated at design time), (b) can be applied by enforcing scheduler-level constraints and (c) do not require any custom hardware and/or architectural support.

In Chapter 4, I propose to incorporate security mechanisms into legacy single core RTS by implementing them as separate, independent *periodic tasks* (called “security tasks”). While there exists some work on reconciling the addition of security mechanisms into RTS [10, 11, 19, 20], they tend to increase execution times for real-time tasks and thus negatively impact timing constraints. Architectural frameworks such as those proposed in prior work [12–15, 18, 21, 22] assume the availability of custom hardware-support for monitoring and hence not suitable for legacy systems. In contrast, I focus on integrating monitoring and detection tasks *without* any perturbation to the real-time scheduling order by using scheduler-level techniques. In order to provide the best protection, the security tasks need to be executed quite often. The challenge then, is to determine the “*right periods*” (i.e., minimum inter-execution time) for the security tasks. Therefore, I introduce a technique that allows the execution of security tasks *opportunistically* with lowest-priority (so that they do not interfere the execution order of real-time tasks), while keeping the best possible periods for the security tasks. I then propose a dual-mode model (named CONTEGO) that allows the security tasks to execute in two different modes: (a) by default security tasks execute opportunistically when the system is deemed to be uncompromised; (b) if an anomaly is suspected, the security tasks may switch to higher priority; (c) the system reverts to “normal” mode if: (i) no anomalous activity is found or (ii) the root cause of the problem is detected and malicious entities are removed. I also propose a metric (called *tightness of monitoring*) that provides us one way to trade-off security with schedulability. I evaluate my propose technique using *time-to-detect* an intrusion as a performance criteria with a view to observing how well the security tasks can perform desired monitoring and detection after period selection.

The techniques proposed in Chapter 4 target single core systems. The use of multicore platforms in real-time CPS is increasingly becoming common since they provide higher performance and better energy efficiency [23]. However, this makes the problem of integrating security mechanisms more complex. Unlike single core systems, integrating security into multicore platforms is more challenging since the designers now have multiple choices for where to allocate the security tasks. For instance, should the engineers: (i) *spread the security tasks to all cores* (in conjunction with the real-time tasks) and if so, how to determine the *task-to-core assignment*? or (iii) *execute them continuously* across any available core? In addition, how the designers can *determine the periods of the security tasks*? Chapter 5 addresses aforementioned issues. In particular, I first develop a low-complexity iterative solution (called HYDRA) that jointly finds the security tasks’ periods and core assignments. The HYDRA mechanism assumes that the security tasks are *statically assigned* across all available cores. I then extend HYDRA with an alternate design mechanism, HYDRA-C, that *can raise the “responsiveness” of monitoring tasks by increasing their frequency of execution*. The key intuition is that if the security tasks are able to execute with as few interruptions as possible (e.g., by moving immediately to an empty core when they are interrupted), then there is much higher chance of successful detection and correspondingly, a much lower chance of successful adversarial action. HYDRA-C provides better monitoring when compared to HYDRA but comes with a cost (in terms of context switch overhead).

The techniques presented in Chapters 4–5 (i.e., CONTEGO, HYDRA, HYDRA-C) are pure software-based (i.e., scheduler-level) solutions since real-time schedulers are considered to be the most important resource management entity and hence is the focus of my work in terms of adding security. I note that my software-based solutions execute both real-time and security tasks in a common platform and hence the security mechanism may collapse if the adversary can compromise the host operating system (OS). In the follow-up work (Chapter 6), I use hardware-assisted trusted modules and execute security checks inside a tamper-resistant platform. Recall from Fig. 1.3 and earlier discussion that, CONTEGO, HYDRA and HYDRA-C abstract application requirements (i.e., security checks are periodic and independent of existing real-time tasks). Since majority cyber-physical applications are largely based on sensing and actuation, any false/spoofed actuation command — i.e., commands that control physical states — can disrupt the normal operation of the plant. In Chapter 6, I leverage hardware-supported security extensions and address application-specific requirements, i.e., *protect physical actuators by checking actuation commands*. I refer to my framework SCATE. Specifically, SCATE uses the concept of trusted execution environments (TEEs) [24] available in commodity processors (e.g., ARM TrustZone [25]) to ensure that security critical execution segments of real-time tasks (e.g., checking of actuation commands) can not be tampered even if the host OS is compromised. Unlike the frameworks proposed in Chapter 4 and Chapter 5 that model detection/monitoring as independent and periodic tasks, security checks in SCATE (i.e., validating actuation commands) are based on the actions of the real-time tasks (i.e., activated when real-time tasks generate actuation commands). I find out such TEE-based checking of actuation commands in real-time applications can lead to significant context switch overhead and a critical task may not comply with its timing requirements. To minimize checking overheads, I therefore develop mechanisms (by using *game theoretical analysis* [26]) that *selectively picks a random subset of actuation commands for checking*. I implement SCATE using off-the-shelf TEE technology (ARM TrustZone) running embedded Linux. I demonstrate the feasibility of SCATE for four real representative systems (viz., ground rover, flight controller, robotic arm and syringe pump) and study the trade-off between security and timing guarantees.

Key Contributions of this Dissertation

- Design-time frameworks (CONTEGO, HYDRA, HYDRA-C) to integrate security tasks into RTS that will allow system designers to improve the security posture without affecting temporal constraints of the existing real-time tasks for both, single core (Chapter 4) and multicore (Chapter 5) platforms.
- A new metric (named *tightness of periodic monitoring*) to measure the effectiveness of such integration (Chapters 4–5).
- A time-aware hardware/software framework (SCATE) to secure COTS-based real-time cyber-physical platforms against attacks that falsify actuation commands (Chapter 6). SCATE

deters attacks with significantly less overheads and also guarantee that it will not violate timing constraints.

Chapters 7–8 discuss the limitations and possible extensions of my proposed techniques. I now start with related research (Chapter 2) and then present my assumptions on attacker’s capabilities (Chapter 3).

CHAPTER 2: RELATED WORK

In this chapter I summarize existing techniques. I have identified some well-defined categories and sub-categories for related real-time scheduling and security research. I now present related work along two fronts: (a) real-time scheduling models (Section 2.1) and (b) real-time CPS security solutions (Section 2.2).

2.1 SCHEDULING AND PERIOD OPTIMIZATION IN RTS

While not in the context of real-time CPS security, there exist some related real-time scheduling frameworks as I present below.

Real-Time Scheduling Frameworks

The system model presented in Chapters 4–5 may be viewed as special case of mixed-criticality systems [27] where the system operates in multiple criticality levels (say “high” for real-time and “low” for security tasks). However, unlike traditional mixed-criticality task model [28] where execution times and periods are vectors of values (see the related survey [27]), I consider a single execution time and period value (i.e., there exists only one criticality level). In mixed criticality systems “low”-criticality tasks are abandoned to ensure timely operation of the “high”-criticality tasks [29]. Abandoning security tasks may not be an option in my context since this will fail to complete security checks in a timely manner. Mixed-criticality scheduling is also different than the problem considered in this dissertation due to the fact that security properties (e.g., adaptive switching depending on runtime behavior or frequent execution of monitoring events for faster detection) are often different than temporal requirements (e.g., satisfying deadline constraints for mixed-criticality tasks). However, the theory and concepts emerged from mixed-criticality systems can also be applied to the real-time security problems to further harden the security posture of future real-time CPS.

The scheduling approaches present in Chapter 5 can be considered as a special case of prior work [30] where each task can bind to any arbitrary number of available cores. For a given period, this prior analysis [30] is pessimistic for the model considered by HYDRA-C (i.e., real-time tasks are partitioned and security tasks can migrate on any core) in a sense that it over-approximates carry-in interference from the tasks bound to single cores (e.g., real-time tasks) and hence results in lower schedulability (i.e., identical to the GLOBAL-TMax scheme in Fig. 5.7). Researchers also propose various semi-partitioned scheduling strategies for fixed-priority RTS [31, 32]. However, these existing work (a) primarily focus on improving schedulability (e.g., by allowing highest priority task to migrate) and (b) are not designed for security requirements in consideration (e.g., minimizing periods and executing security tasks with fewer interruption for faster anomaly detection).

Period Selection in Real-Time/Control Systems

There exists other work [33, 34] in which the authors statically assign the periods for multiple independent control tasks considering control delay as a cost metric. Davare et al. [35] propose to assign task and message periods as well as satisfy end-to-end latency constraints for distributed automotive systems by leveraging schedulability analysis within a convex optimization framework. Previous work use a different model/application scenario (such as controller area networks [36] and/or minimize control delay using utilization-bound tests) and hence can not be directly adapted in my context.

2.2 SECURITY IN REAL-TIME CYBER-PHYSICAL SYSTEMS

Enhancing security in time-critical cyber-physical applications is an active research area (see the related surveys [37, 38]). Security in real-time CPS has been addressed in literature in different contexts — in broader sense this includes (but not limited to) integrating monitoring and intrusion detection mechanisms, protecting communication channels, defending against side-channel attacks, as well as designing hardware/software based architectural solutions.

Securing Communication Messages/Channels

Researchers proposed techniques to secure real-time CPS from man-in-the-middle attacks, where an attacker can compromise communication between system sensors and controllers [39, 40]. The goal is to find trade-offs between control performance and security overheads (e.g., overheads for enforcing data integrity technique such as message authentication codes to prevent the attacks). There has been some work [10, 11] where authors proposed to add security mechanisms (such as encryption) into RTS and considered periodic task scheduling where each task requires a security service whose overhead varies according to the quantifiable level of the service. Unlike my research, all of the aforementioned work require modification of the existing real-time tasks.

Defense Against Side-Channel Attacks

Bao et al. [41] model the behavior of the attacker and introduce a scheduling algorithm. Unlike hard RTS, authors consider a system with aperiodic tasks that have *soft* deadlines. The proposed algorithm provides a trade-off between side-channel information leakage and the number of deadline misses for the real-time tasks. In comparison, I propose to ensure security policies in hard RTS *without* violating temporal constraints and schedulability of the real-time tasks. A state cleanup mechanism is introduced in literature [42] where the authors modify the fixed priority scheduling algorithm to mitigate information leakage through shared resources (e.g., caches). However, this leakage prevention comes at a cost of reduced schedulability. In comparison, I propose to ensure security policies *without* violating temporal constraints and schedulability of the real-time tasks.

Randomization and Architectural Frameworks

Researchers also proposed schedule obfuscation methods [43] to minimize predictability of deterministic RTS scheduler by randomizing the task schedule while providing the necessary real-time guarantees. Unlike my schemes that works at the scheduler-level, there exist architectural frameworks [12–15, 21, 22, 44] that can protect RTS against security vulnerabilities. I highlight that all the aforementioned work require modification to the scheduler or real-time task parameters. They are also not designed to protect against false actuation commands. It is not inconceivable that those architectural frameworks and randomization protocols can be employed on top of my proposed schemes to improve security posture in future real-time CPS.

Trusted Execution and CPS Security

Perhaps the closest line of work to SCATE is PROTC [45] where a monitor in the enclave enforces secure access control policy (given by the control center) for some peripherals of the drone and ensures that only authorized applications can access certain peripherals. Unlike my scheme, PROTC is limited for specific applications (i.e., aerial robotic vehicles), requires a centralized control center to validate/enforce security policies and does not consider real-time requirements. Researchers also proposed anomaly detection approaches for robotic vehicles [17, 46, 47]. These (prior) approaches do not leverage capabilities of TEEs (i.e., are vulnerable if the adversary can compromise the host OS) and do not consider real-time aspects. Researchers also use game-theoretical analysis for (a) general-purpose control systems [48, 49], (b) decision making problems [50] and (c) preventing physical intrusions in CPS [51]. These schemes are not designed to protect the systems against false actuation commands. In addition, they are not timing-aware (i.e., real-time requirements are not considered). To the best of my knowledge, SCATE is the first comprehensive work that introduces the notion of randomized coarse-grain checking using game-theoretical model in order to validate actuation commands in a TEE-enabled real-time CPS.

There also exist large number of research for generic cyber-physical/Internet-of-things-specific embedded systems as well as use of TrustZone to secure traditional embedded/mobile applications (too many to enumerate here, refer to the related surveys [25, 52–55]) — however the consideration of actuation-specific real-time security/scheduling distinguish SCATE from other research.

CHAPTER 3: ASSUMPTIONS ON ADVERSARIAL CAPABILITIES

I now present my adversary model. I consider identical threat models in Chapters 4–5 (see Section 3.1). Section 3.2 presents assumptions on attacker’s capabilities considered in Chapter 6.

3.1 ADVERSARY MODEL FOR CHAPTER 4 AND CHAPTER 5

In Chapters 4–5, I assume that an adversary may destabilize the system by leveraging (known) vulnerabilities. For example, an attacker could compromise the file system (resulting in corrupted information/system log), change the of control/actuation commands or infer side channel information (e.g., user tasks, cache information, thermal profiles) to launch further attacks (say denial of service). While there exists mechanisms (such as Simplex [56, 57]) that guarantee (hardware/software) fault tolerance, I consider the cases where an attacker intentionally induces faults (i.e., adversarial artifacts) that may jeopardize the safety of the system (e.g., results in miss deadlines). My focus is on threats that can be dealt with by *integrating additional security tasks* into the host. The addition of such tasks may necessitate changing the schedule or increasing the execution time of real-time tasks as was the case in earlier work [10, 11, 19, 19, 42, 58, 59]. In this research I consider situations where additional security tasks (see Table 4.1 for related examples) are only allowed to have minimal (Chapter 4) or no impact (Chapter 5) on the schedule of existing real-time tasks and are not allowed to modify real-time parameters. While I use specific intrusion detection mechanisms (e.g., Tripwire [60], a filesystem integrity checking tool) to demonstrate my approach, the ideas presented in Chapter 4–5 are agnostic to the specific monitoring mechanism. The design of integration techniques and the design of the specific security tasks are orthogonal problems. Since I aim to maximize the frequency of execution of security tasks, mechanisms whose performance improves with frequency of execution (e.g., intrusion monitoring and detection tasks or logging/tracing mechanisms) benefit from my model.

3.2 ADVERSARY MODEL FOR CHAPTER 6

My assumptions on adversarial capabilities in Chapter 6 is similar to that considered in prior work [61, 62]. In particular, I assume that an adversary can tamper with the existing control logic to manipulate actuation commands, thus modifying the behavior of a system in undesirable ways (i.e., threaten the safety of the system). I only consider the cases where an adversary’s actions results in the modification of actuation commands. Other classes of attacks such as scheduler side-channel attacks [63, 64], timing anomalies [61, 65] and network-level man-in-the-middle attacks [39, 40] are not within the scope of this work. However, I do discuss how my approach can be extended to other use-cases and mitigate some of those attacks (Chapter 7). I do not make any assumptions as to how an adversary compromises tasks or actuation commands. For instance, bad software

engineering practices leave vulnerabilities in the systems [66]. When the system is developed using a multi-vendor model [20] (where various components are manufactured and integrated by different vendors) malicious code may be injected (say by a less-trusted vendor) during deployment. The adversary may also induce end-users to download modified source code and/or remote access Trojans, say by using social engineering tactics [15]. I do not consider the adversarial cases that require physical access, i.e., the attacker cannot physically control/turn off/damage the actuators or the system.

CHAPTER 4: AN ADAPTIVE FRAMEWORK FOR INTEGRATING SECURITY TASKS IN SINGLE CORE REAL-TIME SYSTEMS

I now assert one part of my dissertation hypothesis (Section 1.1) by presenting a scheduler-level security integration framework. In particular, I focus on integrating security tasks into RTS (especially *legacy* systems) and define a *metric* (named tightness of periodic monitoring) to measure the effectiveness of such integration. I introduce the concept of “*opportunistic execution*” with hierarchical scheduling [67] and then propose a framework, CONTEGO, that allows security tasks to operate in two different *modes*. In CONTEGO, security monitoring tasks (*a*) execute opportunistically with a lowest priority most of the time (i.e., during normal system operation); (*b*) however can adaptively change their mode of operation and execute with a higher priority (for a limited amount of time) if any anomalous behavior is suspected. I evaluate CONTEGO using synthetic workloads as well as with an implementation on a realistic embedded platform (an open-source ARM CPU running real-time Linux). CONTEGO is shown to increase the security posture of RTS without impacting their temporal (and hence, safety) constraints.

4.1 INTRODUCTION

Until recently, cyber-security considerations were an afterthought in the design of real-time CPS. While fault-tolerance has been a design consideration, traditional fault-tolerance techniques that were designed to counter and survive random or accidental faults are not sufficient to deal with cyber-attacks. Given the increasing cyber-attack risks in RTS [3–7, 9], it is essential to integrate resilience against such attacks into the design of RTS. There is also a need to retrofit existing critical RTS with detection, survival and recovery mechanisms. The focus of my research is on integrating or retrofitting security mechanisms into *legacy* RTS. A legacy RTS is one where modification or perturbation of existing real-time tasks’ parameters (such as run-times, periods and task execution orders) is not always feasible. When integrating any security mechanisms into RTS, the designers need to ensure that they do not perturb or impact the real-time functions in any significant way while at the same time provide the necessary level of security. Any security mechanisms that are introduced not only have to co-exist with legacy real-time tasks without violating their real-time and safety constraints but also the *parameters of such legacy tasks cannot be adjusted to accommodate the security tasks*. This creates an apparent tension, especially so in the case of legacy systems — between security requirements (e.g., having enough cycles for effective detection) and the timing and safety requirements. Not only must the security mechanisms work effectively but they must also not interfere with the deadlines of real-time tasks. For instance, any monitoring and detection mechanism has to be designed so that an adversary cannot easily evade it. This may require that the monitoring and detection tasks be run *frequently*. However, the stringent timing constraints in hard RTS introduce additional complexities for the implementation of such cyber-security mechanisms. For instance, the strict deadlines for the completion of periodic hard

Table 4.1: Example of Security Tasks

Security Task	Approach/Tools
File-system checking	Tripwire [60], AIDE [68], etc.
Network packet monitoring	Bro [69], Snort [70], etc.
Hardware event monitoring	Statistical analysis based checks [71] using performance monitors (e.g., perf [72], OProfile [73], etc.)
Application specific checking	Behavior-based detection (see the related work [13–15, 74])

RTS may not allow for frequent execution of security mechanisms. Further, unlike in conventional computing systems, it may not be possible to execute the security mechanisms for arbitrary lengths of time.

In this chapter I aim to improve the security posture of RTS through integration of “security tasks” (i.e., tasks that are specific for intrusion monitoring and detection tasks purposes) into an fixed-priority RTS while ensuring that the existing real-time tasks are not affected by such integration. Security tasks could include protection, detection or response mechanisms, depending on the system requirements — for instance, a sensor correlation task (to detect sensor manipulation) or an anomaly detection task (that checks possible intrusions) [75]. Table 4.1 presents some examples of security tasks that can be integrated into legacy systems (again, this is by no stretch meant to be an exhaustive list). In my experiments I considered intrusion detection as a monitoring mechanism and used Tripwire [60] (a data integrity checking tool) to demonstrate the feasibility of my approach — the ideas presented in this dissertation though apply more broadly to other security mechanisms.

Considerations for Integrating Security Mechanisms. While integrating security tasks into RTS, the following performance criteria need to be considered.

- i) Monitoring Frequency:* In order to provide the best protection, the security tasks need to be executed quite often. On the one hand, if the interval between consecutive monitoring events is too large, the adversary may harm the system (and remain undetected) between two invocations of the security task. On the other hand, if the security tasks are executed very frequently then it may impact the schedulability of the real-time tasks. Herein lies an important trade-off between monitoring frequency and schedulability.
- ii) Responsiveness:* In some circumstances, a security task may need to execute with less interference from higher-priority tasks. For instance, consider the scenario where a security breach is suspected. In such an event the security task may be required to *perform more fine-grained checking instead of waiting for its next periodic slot*. This may result in delayed execution of low-priority, non-critical, real-time tasks. However, the scheduling policy needs

to ensure that the system remains secure without violating real-time constraints for critical, high-priority, real-time tasks.

In this work I first focus on the *monitoring frequency* criterion (Section 4.4.1) and then extend to improve *responsiveness* properties (Section 4.6). In particular, I consider incorporating security mechanisms by implementing them as separate *periodic tasks*. This brings up the challenge of determining the “right periods” (i.e., minimum inter-monitoring time) for the security tasks [76]. For example, consider the integration of an intrusion detection system (IDS) in an existing RTS (for instance Tripwire or AIDE from Table 4.1) that checks integrity of filesystems. For functional correctness the IDS task needs to execute at least once within a certain time period. If such a task is scheduled less frequently or interrupted often before it can complete checking the entire system (say by other, higher priority, real-time tasks), then an adversary could use that opportunity to intrude into the system and modify sensitive file contents before the next invocation of the detection task. In contrast, if the IDS task is executed more frequently, it may interfere the operation of other low priority tasks. My analysis engine takes the real-time task parameters and periodicity requirements of the security tasks and then find the suitable periods for the security tasks without violating timing requirements (refer to Sections 4.5–4.5.2 for a formal model). This is different than criticality-monotonic priority scheduling [77] in mixed-criticality systems [28] where task period and priority ordering are already defined.

In some circumstances a security task may need to complete with less interference (e.g., better responsiveness) from higher-priority real-time tasks. As an example, consider the scenario in which a security breach is suspected and a security task may be required to perform more fine-grained checking instead of waiting for its next execution slot. At the same time, the scheduling policy needs to ensure that the system does not violate real-time constraints for critical, high-priority control tasks. A simple approach to integrate security tasks without perturbing real-time scheduling order is to execute them *opportunistically*, that is, with the lowest priority so that real-time tasks are not affected. However, if the security tasks always execute with lowest priority, they suffer more interference (i.e., preemption from high-priority real-time tasks) and the consequent longer detection time (due to poor response time) will make the security mechanisms less effective. In order to provide *better responsiveness* and increase the effectiveness of monitoring and detection mechanisms, I then propose a multi-mode framework called CONTEGO.¹ For the most part, CONTEGO executes in a *PASSIVE* mode (i.e., with opportunistic execution of intrusion detection tasks). However, CONTEGO will *switch to an ACTIVE mode of operation* to perform additional checks as needed (e.g., fine-grained analysis, used as an example in Section 4.7.2). This *ACTIVE* mode potentially executes with higher priority, while ensuring the timing guarantees of real-time tasks.

Contributions. In this chapter I have the following contributions:

¹In Latin, “Contego“ refers to “*Shield*”. Since my scheme intends to protect RTS against cyber-attacks, I name my framework CONTEGO.

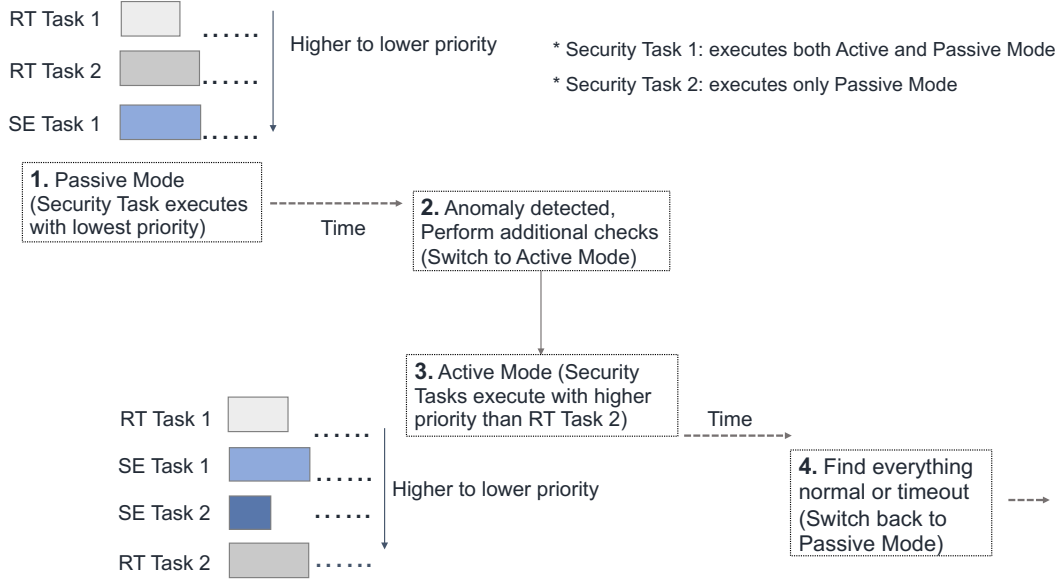


Figure 4.1: CONTEGO: flow of operations depicting the PASSIVE and ACTIVE modes for the security tasks.

- I introduce CONTEGO, an extensible framework to integrate security tasks into legacy RTS (Section 4.2).
- CONTEGO allows the security tasks to execute with minimal perturbation of the scheduling order of the real-time tasks while guaranteeing their timing constraints (Sections 4.5–4.6). The proposed method can adapt to changes due to malicious activities by switching its mode of operation.
- I propose a *metric* (named “tightness of monitoring”) to measure the security posture of the system in terms of frequency of execution (Section 4.4).

I also evaluate the schedulability and security of the proposed approach using a range of synthetic task sets and a prototype implementation on an ARM-based development board with real-time Linux (Section 4.7). I now start with a brief overview of CONTEGO.

4.2 OVERVIEW OF CONTEGO

As illustrated in Fig. 4.1, CONTEGO improves the security posture of the system (that contains a set of real-time tasks) by integrating additional security tasks and allowing them to execute in two different *modes* (viz., PASSIVE and ACTIVE). I highlight that rather than designing specific intrusion detection tasks that target specific attack behaviors, the generic framework proposed in this chapter allows one to integrate a given security mechanism (referred to as *security tasks*) into the system without perturbing the system parameters (e.g., period and task execution execution

order). If the system is deemed to be clean (i.e., not compromised), security tasks can execute *opportunistically*² (e.g., when other real-time tasks are not running). However if any anomaly or unusual behavior is suspected, the security policy may switch to ACTIVE mode (e.g., more fine-grained checking or response) and execute with *higher priority* for a *limited amount of time* (since our goal is to ensure security with minimum perturbation of the scheduling order of the real-time tasks). The security tasks may go back to normal (e.g., PASSIVE) mode if:

- No anomalous activity is found within a predefined time duration, say T^{AC} ; or
- The intrusion is detected and malicious entities are removed (or an alarm triggered if human intervention is required).

Although I allow the security tasks to execute with higher priority than some of the real-time tasks in ACTIVE mode, the proposed framework ensures that the timeliness constraints (e.g., deadlines) for *all* of the real-time tasks are always satisfied in *both* modes. By using this strategy, CONTEGO not only enables *compatibility with legacy systems* (i.e., in normal situation real-time scheduling order is not perturbed), but also provides *flexibility to promptly deal with anomalous behaviors* (i.e., the security tasks are promoted to higher priority so that they can experience less preemption and achieve better response times).

4.3 SYSTEM MODEL

4.3.1 Real-Time Tasks

In this paper I consider the widely used fixed-priority sporadic task model [78]. In particular, I consider a uniprocessor system consisting of m fixed-priority sporadic real-time tasks $\Gamma_R = \{\tau_1, \tau_2, \dots, \tau_m\}$. Each real-time task $\tau_j \in \Gamma_R$ is characterized by (C_j, T_j, D_j) , where C_j is the worst-case execution time (WCET) [79], T_j is the minimum inter-arrival time (or period) between successive releases and D_j is the relative deadline. We assume that priorities are distinct and assigned according to the rate monotonic (RM) [80] order (i.e., short task period implies higher priority).

The processor utilization of τ_j is defined as $U_j = \frac{C_j}{T_j}$. Let $hp_R(\tau_j)$ and $lp_R(\tau_j)$ denote the sets of real-time tasks that have higher and lower priority than τ_j , respectively. We assume that the real-time task-set Γ_R is *schedulable* by a fixed-priority preemptive scheduling algorithm. Therefore, the worst-case response time w_i is less than or equal to the deadline D_i and the following inequality is satisfied for all tasks $\tau_j \in \Gamma_R$: $w_j \leq D_j$, where $w_j = w_j^{k+1} = w_j^k$ is obtained by the following recurrence relation [81]:

$$w_j^0 = C_j, \quad w_j^{k+1} = C_j + \sum_{\tau_h \in hp_R(\tau_j)} \left\lceil \frac{w_j^k}{T_h} \right\rceil C_h. \quad (4.1)$$

²Which is also the default mode of operation.

In Eq. (4.1), $\sum_{\tau_h \in hp_R(\tau_j)} \left\lceil \frac{w_j^k}{T_h} \right\rceil C_h$ is the worst-case interference to τ_j due to preemption by the tasks with higher priority than τ_j (i.e., $hp_R(\tau_j)$). The recurrence will have a solution if $w_j^{k+1} = w_j^k$ for some k .

4.3.2 Security Tasks

I model PASSIVE and ACTIVE mode security tasks as independent *periodic tasks*. The PASSIVE and ACTIVE mode tasks are denoted by the sets $\Gamma_S^{pa} = \{\tau_1, \tau_2, \dots, \tau_{n_p}\}$ and $\Gamma_S^{ac} = \{\tau_1, \tau_2, \dots, \tau_{n_a}\}$, respectively. I assume that security tasks in both modes follow RM priority order. Each security task $\tau_i \in \{\Gamma_S^{pa} \cup \Gamma_S^{ac}\}$ is characterized by the tuple $(C_i, T_i^{des}, T_i^{max}, \omega_i)$, where C_i is the WCET, T_i^{des} is the most desired period between successive releases (i.e., $F_i^{des} = \frac{1}{T_i^{des}}$ is the desired execution frequency of a security routine) and T_i^{max} is the maximum allowable period beyond which security checking by τ_i may not be effective. The parameter $\omega_i > 0$ is a designer-provided weighting factor that may reflect the criticality of the security task³ τ_i . Critical security tasks would have larger ω_i . The security tasks have implicit deadlines, i.e., $D_i = T_i, \forall \tau_i$ that implies security tasks should complete before their next monitoring instance. I do not make any specific assumptions about the security tasks in different modes. For instance, both PASSIVE and ACTIVE mode task-sets may contain completely different sets of tasks (i.e., $\{\Gamma_S^{pa} \cap \Gamma_S^{ac}\} = \emptyset$) or may contain (partially) identical tasks with different parameters (i.e., period and/or criticality requirements).

In PASSIVE mode, security tasks are executed with *lower priority* than the real-time tasks. Hence the security tasks do not have any impact on real-time tasks and cannot perturb the real-time scheduling order. In ACTIVE mode, I allow the security tasks to execute with a priority higher than that of certain low priority real-time tasks. This provides us with a trade-off mechanism between security (e.g., responsiveness) and system constraints (e.g., scheduling order of real-time tasks). Since the task priorities are distinct, there are m priority-levels for real-time tasks (indexed from 0 to $m - 1$ where level 0 is the highest priority). Among the m priority-levels, we assume that ACTIVE mode security tasks can execute with a priority-level up to l_S ($0 < l_S \leq m$), $l_S \in \mathbb{Z}$. Although any period T_i within the range $T_i^{des} \leq T_i \leq T_i^{max}$ is acceptable for PASSIVE (e.g., $\tau_i \in \Gamma_S^{pa}$) and ACTIVE (e.g., $\tau_i \in \Gamma_S^{ac}$) mode security tasks, the actual period T_i is not known a priori. Furthermore, for ACTIVE mode security tasks (e.g., $\tau_i \in \Gamma_S^{ac}$), we need to find out the suitable priority level $l \in [l_S, m]$. Therefore our goal is to find the *suitable period* (for both PASSIVE and ACTIVE mode security tasks) as well as the *priority-level* (for ACTIVE mode security tasks) that achieve the best trade-off between schedulability and defense against security breaches without violating the real-time constraints.

³As an example, the default configuration of Tripwire [60], an intrusion detection system (IDS) for Linux that I use as case study in Section 4.7.2, has different criticality levels (viz., weights), e.g., *High* (for scanning files that are significant points of vulnerability), *Medium* (for non-critical files that are of significant security impact) and so forth.

4.4 PERIOD ADAPTATION

As already mentioned, one fundamental problem in integrating security tasks is to determine *which* security tasks will be running *when*. This is different when compared to scheduling traditional real-time tasks since the real-time task parameters (e.g., periods) are often derived from physical system properties and cannot be adjusted due to control/application requirements. One may wonder why I cannot assign the desired period (e.g., $T_i = T_i^{des}$) in both PASSIVE and ACTIVE modes and set the ACTIVE mode priority level as $l = l_S$ so that the security tasks can always execute with the desired frequency (i.e., $F_i^{des} = \frac{1}{T_i^{des}}$) and experience less interference (e.g., preemption) from real-time tasks. However, since my goal is to integrate security mechanisms in legacy systems with minimal⁴ or no perturbation, setting $T_i = T_i^{des}$, $\forall \tau_i$ in either or both mode(s) may significantly perturb the real-time scheduling order. If the schedulability of the system is not analyzed after the perturbation, some (or all) of the real-time tasks may miss their deadlines and thus the main safety requirements of the system will be threatened. The same argument is also true for ACTIVE mode if I set $l = l_S$ (or arbitrarily from the range $[l_S, m]$) and do not perform schedulability analysis carefully.

Tightness of the Monitoring

Recall that the actual period as well as the priority-levels of the security tasks are unknown and we need to *adapt* the periods within acceptable ranges. I measure the security of the system by means of *achievable periodic monitoring*. Let T_i be the period of the security task $\tau_i \in \{\Gamma_S^{pa} \cup \Gamma_S^{ac}\}$ that needs to be determined. My goal is to minimize the gap between the achievable period T_i and the desired period T_i^{des} and therefore I use the the following metric:

$$\eta_i = \frac{T_i^{des}}{T_i}, \quad (4.2)$$

that denotes the *tightness* of the frequency of periodic monitoring for the security task τ_i . Thus $\eta^{pa} = \sum_{\tau_i \in \Gamma_S^{pa}} \omega_i \eta_i$ and $\eta^{ac} = \sum_{\tau_i \in \Gamma_S^{ac}} \omega_i \eta_i$ denote the *cumulative tightness* of the achievable periodic monitoring for PASSIVE and ACTIVE mode, respectively. This monitoring frequency metric, provides for instance, one way to trade-off security with schedulability. Recall that if the interval between consecutive monitoring events is too large, the adversary may remain undetected and harm the system between two invocations of the security task. Again, a very frequent execution of security tasks may impact the schedulability of the real-time tasks. This metric $\eta^{(\cdot)}$ will allow us to execute the security routines with a frequency closer to the desired one while respecting the temporal constraints of the other real-time tasks.

⁴In ACTIVE mode CONTEGO does not introduce any timing violations for the real-time tasks, but their execution might be delayed due to interference from high-priority security tasks (i.e., the tasks with priority-level $l \in [l_S, m]$).

4.4.1 Problem Overview

One may wonder why we cannot schedule the security tasks in the same way that the existing real-time tasks are scheduled. For instance, a simple approach to integrating security tasks in PASSIVE mode without perturbing real-time scheduling order is to execute security tasks at a *lower priority* than all real-time tasks. Hence, the security routines will be executing only during slack times when no other higher-priority real-time tasks are running. Likewise, in ACTIVE mode, security tasks can be executed at a lower priority than more critical, high-priority real-time tasks. Hence, the security tasks will only be executing when other real-time tasks with priority-levels higher than l_S are not running.

When both real-time and security tasks follow RM priority order, we can formulate a nonlinear optimization problem for PASSIVE mode with the following constraints that maximizes the cumulative tightness of the frequency of periodic monitoring:

$$\text{(P4.1)} \quad \max_{\mathbf{T}^{pa}} \eta^{pa} \quad (4.3)$$

$$\text{Subject to: } \sum_{\tau_i \in \Gamma_S^{pa}} \frac{C_i}{T_i} \leq (m + n_p)(2^{\frac{1}{m+n_p}} - 1) - \sum_{\tau_j \in \Gamma_R} \frac{C_j}{T_j} \quad (4.4)$$

$$T_i \geq \max_{\tau_j \in \Gamma_R} T_j \quad \forall \tau_i \in \Gamma_S^{pa} \quad (4.5)$$

$$T_i^{des} \leq T_i \leq T_i^{max} \quad \forall \tau_i \in \Gamma_S^{pa} \quad (4.6)$$

where $\mathbf{T}^{pa} = [T_1, T_2, \dots, T_{n_p}]^T$ is the optimization variable for PASSIVE mode that needs to be determined. The constraint in Eq. (4.4) ensures that the utilization of the security tasks are within the remaining RM utilization bound [80]. The RM priority order for real-time and security tasks is ensured by the constraints in Eq. (4.5), while Eq. (4.6) ensures the restrictions on periodic monitoring.

Recall that in ACTIVE mode, I allow the security tasks to execute when the real-time tasks with priority-levels higher than l_S are not running. Hence, to ensure the RM priority order in ACTIVE mode, we need to modify the constraints in Eq. (4.5) as follows:

$$T_i \geq \max_{\tau_j \in \Gamma_{R_{hp}(l_S)}} T_j, \quad \forall \tau_i \in \Gamma_S^{ac} \quad (4.7)$$

where $\Gamma_{R_{hp}(l_S)}$ represents the set of real-time tasks that are higher priority than level l_S . In addition, the constraints in Eq. (4.4) and Eq. (4.6) also need to be updated to consider ACTIVE mode task-sets (e.g., Γ_S^{ac}) and the number of active mode security tasks (n_a). Thus for ACTIVE mode we can formulate an optimization problem similar to that of **P4.1** with the objective function: $\max_{\mathbf{T}^{ac}} \eta^{ac}$, where $\mathbf{T}^{ac} = [T_1, T_2, \dots, T_{n_a}]^T$ is the ACTIVE mode optimization variable.

Although it is non-trivial to solve the above non-linear non-convex optimization problem in its current form, it is possible to transform the above formulations into a convex optimization problem

using an approach similar to that presented in this chapter. However, one of the limitations of the above approach is that the overall system utilization is limited by the RM bound which has the theoretical upper bound of processor utilization only about $\lim_{n \rightarrow \infty} n(2^{\frac{1}{n}} - 1) = \ln 2 \approx 69.31\%$ [80], where n is the total number of tasks under consideration. Further, the security tasks' periods need to satisfy the constraints in Eq. (4.5) and Eq. (4.7) (for PASSIVE and ACTIVE modes, respectively) to follow RM priority order. In addition, instead of focusing only on optimizing the periods of the security tasks, CONTEGO aims to provide an adaptive framework that can achieve other security aspects (viz., responsiveness). Hence, instead of simply running security tasks by themselves in idle-times or within predefined priority ranges, I propose using a “*server*” [67] to execute the security tasks. With this approach, for instance, if better responsiveness is desired from security mechanisms, we could increase the priority of the server and allow the server to execute until the security task finishes its desired checking. Not only will the server abstraction allow me to provide better isolation between real-time and security tasks; but it also enables me to integrate responsiveness properties as I discuss in the following.

4.5 THE SECURITY SERVER

The server [67] is an abstraction that provides execution time to the security tasks according to a predefined scheduling algorithm. My proposed security server is characterized by the *capacity* Q and *replenishment period* P and works as follows. The server is executed with lowest-priority in PASSIVE mode. However, in ACTIVE mode, the server can switch to any allowable priority-level⁵ within the range $[l_S, m]$. If any security task is activated at time t , then the server is activated with capacity Q and the next replenishment time is set as $t + P$. When the server is scheduled, it executes the security tasks according to its own scheduling policy. In this work I assume that the server schedules the security tasks using fixed-priority RM scheduling. When a security task executes, the current available capacity is decremented accordingly. The server can be preempted by the scheduler to service real-time tasks. When the server is preempted, the currently available capacity is not decremented. If the available capacity becomes zero and some security task has not yet finished, then the server is suspended until its next replenishment time (t'). At time t' , the server is recharged to its full capacity Q , the next replenishment time is set as $t' + P$, and the server is executed again. When the last security task has finished executing and there is no other pending task in the server, the server will be suspended. Also, the server will become inactive if there are no security tasks ready to execute.

4.5.1 Reformulation of the Period Adaptation Problem using Servers

When security tasks execute within the server, we need to modify the constraints in the period adaption problem considering the server parameters Q and P . In the following I briefly discuss

⁵Calculation of the server priority-level is described in Section 4.6.

how to customize the period adaptation problem with the inclusion of the server.

Let me use $UB_{S(Q,P),\Gamma}$ to denote the utilization bound for the set of tasks Γ executing within the server. When the smallest period of the task is greater than or equal to $3P - 2Q$, it has been shown [82] that the upper bound of the utilization factor for the security tasks is given by

$$UB_{S(Q,P),\Gamma} = n \left[\left(\frac{3 - \frac{Q}{P}}{3 - 2\frac{Q}{P}} \right)^{\frac{1}{n}} - 1 \right], \text{ where } n \text{ is number of tasks in the set } \Gamma.$$

Thus with the inclusion of the server in PASSIVE mode, I now modify the constraints in Eqs. (4.4) and (4.5) as follows:

$$\sum_{\tau_i \in \Gamma_S^{pa}} \frac{C_i}{T_i} \leq n_p \left[\left(\frac{3 - \frac{Q^{pa}}{P^{pa}}}{3 - 2\frac{Q^{pa}}{P^{pa}}} \right)^{\frac{1}{n_p}} - 1 \right] \quad (4.8)$$

$$T_i \geq 3P^{pa} - 2Q^{pa}, \quad \forall \tau_i \in \Gamma_S^{pa}. \quad (4.9)$$

Therefore, selection of the periods for security tasks in PASSIVE mode is a nonlinear constrained optimization problem that can be formulated as follows:

$$\text{(P4.2)} \quad \max_{\mathbf{T}^{pa}} \sum_{\tau_i \in \Gamma_S^{pa}} \omega_i \frac{T_i^{des}}{T_i} \quad (4.10)$$

Subject to: Eqs. (4.8), (4.9), (4.6).

where Q^{pa} and P^{pa} are the server capacity and replenishment period in PASSIVE mode, respectively.

Similarly, in ACTIVE mode, the period adaptation problem can be reformulated as follows:

$$\text{(P4.3)} \quad \max_{\mathbf{T}^{ac}} \sum_{\tau_i \in \Gamma_S^{ac}} \omega_i \frac{T_i^{des}}{T_i} \quad (4.11)$$

$$\text{Subject to: } \sum_{\tau_i \in \Gamma_S^{ac}} \frac{C_i}{T_i} \leq n_a \left[\left(\frac{3 - \frac{Q^{ac}}{P^{ac}}}{3 - 2\frac{Q^{ac}}{P^{ac}}} \right)^{\frac{1}{n_a}} - 1 \right] \quad (4.12)$$

$$T_i \geq 3P^{ac} - 2Q^{ac} \quad \forall \tau_i \in \Gamma_S^{ac} \quad (4.13)$$

$$T_i^{des} \leq T_i \leq T_i^{max} \quad \forall \tau_i \in \Gamma_S^{ac} \quad (4.14)$$

where Q^{ac} and P^{ac} are the server capacity and replenishment period in ACTIVE mode, respectively.

4.5.2 Selection of the Server Parameters

The period adaptation problem illustrated in Section 4.5.1 is derived based on a given set of server parameters, e.g., $(Q^{(\cdot)}, P^{(\cdot)})$. However, a fundamental problem is to find a suitable pair of server capacity $Q^{(\cdot)}$ and replenishment period $P^{(\cdot)}$ that respects the real-time constraints of the

tasks in the system. My approach to selecting the server parameters in PASSIVE and ACTIVE mode is described below.

Parameter Selection in Passive Mode

Recall that in PASSIVE mode, the server will execute with the lowest priority to have compatibility with existing real-time tasks. Since the security tasks execute within the server, we need to ensure the following two constraints:

- *The server is schedulable*: that is the server's capacity and interference from higher priority real-time tasks are less than the replenishment period; and
- *The security tasks are schedulable*: the minimum *supply* by the server to the security tasks is greater than the worst-case workload generated by the security tasks.

Note that since the server is running with lowest priority, the real-time constraints (e.g., $w_j \leq D_j, \forall \tau_j \in \Gamma_R$) and the task execution order are not affected in the PASSIVE mode. Based on the above two constraints, I illustrate an approach for determining the server parameters by formulating it as a *constraint optimization problem*.

The security server is referred to as *schedulable* if the worst-case response time of the server does not exceed its replenishment period [67]. Thus, following an approach similar to ones in earlier work [83, 84], the *server schedulability constraint* can be represented as follows:

$$Q^{pa} + \Delta_{S^{pa}} \leq P^{pa} \quad (4.15)$$

where $\Delta_{S^{pa}} = \sum_{\tau_h \in hp_R(\tau_{S^{pa}})} \left(\frac{P^{pa}}{T_h} + 1 \right) C_h$ is the worst-case interference experienced by the server when preempted by the higher priority real-time tasks. In the above equation, the set of real-time tasks with higher priority than the server (i.e., $hp_R(\tau_S^{pa}) = \Gamma_R$) is fixed.

Let me use $hp_S^{pa}(\tau_i)$ to denote the set of PASSIVE mode security tasks that are higher priority than $\tau_i \in \Gamma_S^{pa}$. To ensure schedulability of the security tasks, we can derive the *minimum supply* of the server delivered to the security tasks by using the periodic resource model from the literature [83–85]. In particular, the constraints on the server supply to ensure *schedulability of the security tasks* can be expressed as (refer to Appendix A.1 for formal derivations):

$$\frac{Q^{pa}}{P^{pa}} [T_i - (P^{pa} - Q^{pa}) - \Delta_{S^{pa}}] \geq I_i^{pa}, \quad \forall \tau_i \in \Gamma_S^{pa} \quad (4.16)$$

where $I_i^{pa} = C_i + \sum_{\tau_h \in hp_S^{pa}(\tau_i)} \left\lceil \frac{T_i}{T_h} \right\rceil C_h$ is the worst-case workload generated by the security task τ_i and $hp_S^{pa}(\tau_i)$ during the time interval of T_i . This workload is a constant for a given input.

Since I need to ensure maximal processor utilization for the security tasks without violating the real-time constraints of the system, I define the following objective function: $\max_{Q^{pa}, P^{pa}} \frac{Q^{pa}}{P^{pa}}$. With this

objective function and the constraints in Eqs. (4.15)–(4.16), the PASSIVE mode server parameter selection problem can be formulated as follows:

$$\text{(P4.4)} \quad \max_{Q^{pa}, P^{pa}} \frac{Q^{pa}}{P^{pa}} \quad (4.17)$$

Subject to: Eqs. (4.15), (4.16)

where server parameters Q^{pa} and P^{pa} are the optimization variables.

Parameter Selection in Active Mode

In ACTIVE mode, the security server is *no longer the lowest priority task*. Since the server can execute with priority l_S , there could be up to $m - l_S$ low priority real-time tasks than that of the server. Thus we need to ensure the schedulability of the real-time tasks that are executing with a priority lower than the server. Hence, in addition to the constraints described in Section 4.5.2 (i.e., Eqs. (4.15)–(4.16)), we need to consider the following:

- *The real-time tasks with lower priority than the server are schedulable:* that is, the interferences from the server and other higher priority real-time tasks do not violate the deadlines for these low-priority tasks.

I therefore define the following constraints to ensure the *schedulability of the low-priority real-time tasks*:

$$C_j + \sum_{\tau_h \in hp_R(\tau_j)} \left\lceil \frac{D_j}{T_h} \right\rceil C_h + \left(\frac{D_j}{P^{ac}} + 1 \right) Q^{ac} \leq D_j, \quad \forall \tau_j \in lp_R(\tau_S^{ac}) \quad (4.18)$$

where $\sum_{\tau_h \in hp_R(\tau_j)} \left\lceil \frac{D_j}{T_h} \right\rceil C_h$ is the interference experienced by τ_j from other real-time tasks and $\left(\frac{D_j}{P^{ac}} + 1 \right) Q^{ac}$ is the worst-case interference caused to τ_j by the server in ACTIVE mode. As illustrated in Section 4.6, I iterate through the allowable priority ranges (e.g., $[l_S, m]$) to find the server priority in ACTIVE mode. Note that for a given priority-level, the set of tasks $lp(\tau_S^{ac})$ is predefined. Thus the only variables for the constraints in Eq. (4.18) are the server capacity Q^{ac} and replenishment period P^{ac} .

Let me use $hp_S^{ac}(\tau_i)$ to denote the set of ACTIVE mode security tasks that are higher priority than $\tau_i \in \Gamma_S^{ac}$. Just as in **P4.4** I now formulate the ACTIVE mode parameter selection problem as follows:

$$(P4.5) \quad \max_{Q^{ac}, P^{ac}} \frac{Q^{ac}}{P^{ac}}, \quad (4.19)$$

Subject to: Eq. (4.18) and

$$Q^{ac} + \sum_{\tau_h \in hp_R(\tau_{S^{ac}})} \left(\frac{P^{ac}}{T_h} + 1 \right) C_h \leq P^{ac} \quad (4.20)$$

$$\frac{Q^{ac}}{P^{ac}} [T_i - (P^{ac} - Q^{ac}) - \Delta_{S^{ac}}] \geq I_i^{ac} \quad \forall \tau_i \in \Gamma_S^{ac} \quad (4.21)$$

where the set of real-time tasks with higher priority than the server (i.e., $hp_R(\tau_S^{ac}) \subset \Gamma_R$) is a constant for a given priority-level and $I_i^{ac} = C_i + \sum_{\tau_h \in hp_S^{ac}(\tau_i)} \left\lceil \frac{T_i}{T_h} \right\rceil C_h$ is the worst-case workload generated by the security task τ_i and $hp_S^{ac}(\tau_i)$. Note that the schedulability of the higher priority real-time tasks (i.e., $\forall \tau_j \in hp_R(\tau_S^{ac})$) is already ensured by definition.

Remark 4.1. *The formulation of the period adaptation and server parameter selection problems are nonlinear constraint optimization problems and are nontrivial to solve in their current form. However, these problems can be transformed into a geometric programming (GP) [86] problem. In addition, it is also possible to reformulate the non-convex GP representation into equivalent convex form that can be solved using known algorithms such as the interior point [87, Ch. 11] method. For details of this reformulation, I refer the readers to Appendix A.2.*

4.5.3 Discussion on Mode Switching

As mentioned earlier, by default, CONTEGO operates in PASSIVE mode (i.e., execute opportunistically). However, when a malicious activity is suspected, a PASSIVE-to-ACTIVE mode change request will be issued. Similarly, an ACTIVE-to-PASSIVE mode change request will be placed if the system seems clean after fine-grained checking, or a malicious entity is found and removed. In steady-state (e.g., when security tasks are executing in PASSIVE or ACTIVE mode), the schedulability of the real-time tasks is already guaranteed by the analysis presented in Section 4.5.2.

When CONTEGO switches from PASSIVE mode to ACTIVE mode, the schedulability of real-time tasks will not be affected. The reason is that *all* the real-time tasks are higher priority than the security tasks in PASSIVE mode and hence do not suffer any additional interference from security tasks during mode change. Therefore, the schedulability of real-time tasks during PASSIVE-to-ACTIVE mode switching is already covered by steady-state analysis (Section 4.5.2).

During ACTIVE-to-PASSIVE mode switching, observe that schedulability of the real-time tasks that have a priority higher than the server (i.e., $hp_R(\tau_S^{ac})$) is not affected. When the mode switch request is issued, the ACTIVE mode server (and the security tasks) stop execution and the control is then switched to the lowest priority PASSIVE mode server. Note that the constraints in Eq. (4.18) that ensures the schedulability of the low-priority real-time tasks already captures the worst-case interference introduced by the server. Hence the server will not impose any more interference (even if the mode switch is performed in the middle of the execution of a busy interval) on the

low-priority real-time tasks than what I have calculated in the steady-state analysis (Section 4.5.2). Therefore if both the PASSIVE and ACTIVE modes task-sets are schedulable, the system will also be schedulable with mode changes.

4.6 ALGORITHM

I develop a simple scheme to obtain the security task’s period (for both PASSIVE and ACTIVE mode) and priority-level (for ACTIVE mode). The overall algorithm, Algorithm 4.1, works as follows.

To find the PASSIVE mode parameters, I initialize the security task’s period with the desired period and solve the server parameter selection problem **P4.4** (Lines 10–11). If there exists a solution (e.g., the constraints are satisfied), I then obtain the periods of the security tasks by solving **P4.2** (Line 13). In the event that neither of these optimization problems returns a solution, I report the task-set as unschedulable (Line 20), since it is not possible to execute security tasks opportunistically without violating real-time constraints.

To select ACTIVE mode parameters, the algorithm iterates through each of the acceptable priority-levels $[l_s, m]$ and tries to obtain the periods that maximize tightness for periodic monitoring without violating the real-time constraints (Lines 26–36). If there exists a solution (e.g., constraints in **P4.5** and **P4.3** are mutually consistent), I store the solution in a candidate list. The algorithm then finds the best priority-level from the candidate solution sets that provides the maximum tightness (Line 39). In the event that no candidate solutions are found for any of the allowable priority ranges, the algorithm reports the task-set as unschedulable.

If *both* the PASSIVE and ACTIVE mode tasks are schedulable, then Algorithm 4.1 returns the corresponding periods and the ACTIVE mode priority-level (Line 4). Otherwise, the system is considered as unschedulable (Line 7) since it is not possible to integrate security tasks with desired requirements. This unschedulability result hints that the designers of the system should update system parameters (e.g., the number of security tasks, desired and maximum allowable periods of the security tasks, periods of the real-time tasks, if permissible) in order to integrate security mechanisms.

4.7 EVALUATION

I evaluate CONTEGO with randomly generated synthetic workloads (Section 4.7.1) as well as a proof-of-concept implementation on an ARM-based embedded development board and real-time Linux (Section 4.7.2). My implementation is available in a public repository [88].

Algorithm 4.1: Feasibility Checking and Parameter Selection

Input: Set of real-time tasks, Γ_R , PASSIVE and ACTIVE mode security tasks Γ_S^{pa} and Γ_S^{ac} , allowable priority ranges $[l_S, m]$

Output: The tuple $\{l^*, \mathbf{T}^{pa}, Q^{pa}, P^{pa}, \mathbf{T}^{ac}, Q^{ac}, P^{ac}\}$, e.g., ACTIVE mode server priority-level, ACTIVE and PASSIVE mode periods of the security tasks and ACTIVE and PASSIVE mode server parameters if the task-set is schedulable; **Unschedulable** otherwise

- 1: Obtain PASSIVE and ACTIVE mode parameters using the functions $\text{PASSIVEMODEPARAMSELECTION}(\Gamma_R, \Gamma_S^{pa})$ and $\text{ACTIVEMODEPARAMSELECTION}(\Gamma_R, \Gamma_S^{ac}, l_S)$
- 2: **if** Solution Found in BOTH Modes **then**
- 3: **return** $\{l^*, \mathbf{T}^{pa}, Q^{pa}, P^{pa}, \mathbf{T}^{ac}, Q^{ac}, P^{ac}\}$ */* return the parameters */*
- 4: **else**
- 5: **return** **Unschedulable** */* not possible to integrate security tasks in the system */*
- 6: **end if**

- 7: **function** $\text{PASSIVEMODEPARAMSELECTION}(\Gamma_R, \Gamma_S^{pa})$
- 8: Initialize PASSIVE mode period $T_i := T_i^{des}, \forall \tau_i \in \Gamma_S^{pa}$
- 9: Solve **P4.4** to obtain server parameters
- 10: **if** SolutionFound **then**
- 11: Solve **P4.2** to obtain security periods
- 12: **if** SolutionFound **then**
- 13: */* return the parameters */*
- 14: **return** $\mathbf{T}^{pa}, Q^{pa}, P^{pa}$ where Q^{pa}, P^{pa} and \mathbf{T}^{pa} are the solutions obtained by **P4.4** and **P4.2**
- 15: **end if**
- 16: **else**
- 17: **return** **Unschedulable** */* unable to integrate PASSIVE mode security tasks */*
- 18: **end if**
- 19: **end function**

- 20: **function** $\text{ACTIVEMODEPARAMSELECTION}(\Gamma_R, \Gamma_S^{ac}, l_S)$
- 21: Schedulable := **false**
- 22: Initialize ACTIVE mode security task's period $\mathbf{T}(l')_{\forall l' \in [l_S, m]} := [T_i^{des}]_{\forall \tau_i \in \Gamma_S^{ac}}$
- 23: **for each** priority level $l' \in [l_S, m]$ **do**
- 24: Solve **P4.5** to obtain server parameters
- 25: **if** SolutionFound **then**
- 26: Solve **P4.3** to obtain security periods
- 27: **if** SolutionFound **then**
- 28: */* store the parameters for priority level l' where Q^*, P^* and \mathbf{T}^* are the solutions obtained by **P4.5** and **P4.3** */*
- 29: $Q(l') := Q^*, P(l') := P^*, \mathbf{T}(l') := \mathbf{T}^*$
- 30: Schedulable := **true**
- 31: **end if**
- 32: **end if**
- 33: **end for**
- 34: */* obtain the parameters that provide best metric */*
- 35: **if** Schedulable **then**
- 36: Find the priority-level l^* from the solution vector $\mathbf{T}(l')_{\forall l' \in [l_S, m]}$ tasks at l' is schedulable that gives the maximum cumulative tightness $\eta^{ac} = \sum_{\tau_i \in \Gamma_S^{ac}} \eta_i$
- 37: Set $\mathbf{T}^{ac} := \mathbf{T}(l^*), Q^{ac} := Q(l^*), P^{ac} := P(l^*)$
- 38: */* return the parameters */*
- 39: **return** $l^*, \mathbf{T}^{ac}, Q^{ac}, P^{ac}$
- 40: **else**
- 41: **return** **Unschedulable** */* unable to integrate ACTIVE mode security tasks */*
- 42: **end if**
- 43: **end function**

4.7.1 Experiment with Synthetic Task-sets

Simulation Setup

In order to generate task-sets with an even distribution of tasks, I grouped the real-time and security task-sets by base-utilization from $[0.01 + 0.1 \cdot i, 0.1 + 0.1 \cdot i]$, where $i \in \mathbb{Z} \wedge 0 \leq i \leq 9$. Each

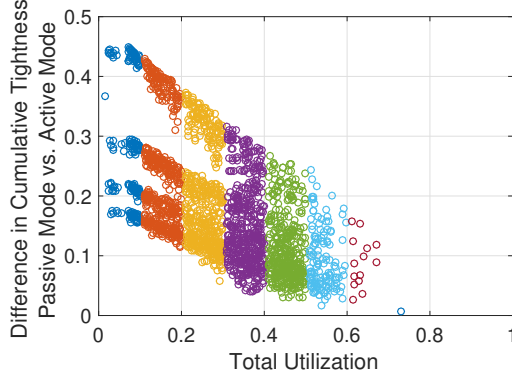


Figure 4.2: PASSIVE mode vs. ACTIVE mode: difference in cumulative tightness of achievable periodic monitoring, $\eta^{av} - \eta^{pa}$. Non-zero difference indicates that the ACTIVE mode tasks achieve better tightness than PASSIVE mode tasks. Task-sets from different base-utilization groups are represented by different colors. Each of the data points represents schedulable task-sets.

utilization group contained 500 task-sets. In other words, a total of 5000 task-sets were tested for each of the experiments. The utilization of the real-time and security tasks were generated by the UUniFast [89] algorithm and I used GPLAB [90] to solve the optimization problems.

I used the parameters similar to those used in earlier research [19, 83]. In particular, each task-set instance contained $[3, 10]$ real-time and $[2, 5]$ security tasks in each of the modes. Each real-time task $\tau_j \in \Gamma_R$ had a period $T_j \in [10 \text{ ms}, 100 \text{ ms}]$ and we assumed $l_S = [0.4m]$. The desired periods for the security tasks $\forall \tau_i \in \{\Gamma_S^{pa} \cup \Gamma_S^{ac}\}$ were selected from $[1000 \text{ ms}, 3000 \text{ ms}]$ and the maximum allowable period was assumed to be $T_i^{max} = 10T_i^{des}$. I considered $\omega_i = 1, \forall \tau_i \in \{\Gamma_S^{pa} \cup \Gamma_S^{ac}\}$ and the total utilization of the security tasks was assumed to be no more than 30% of the real-time tasks.

Results

Impact on Cumulative Tightness. In Fig. 4.2 we can see the difference in the tightnesses of the periodic monitoring obtained by PASSIVE and ACTIVE mode (i.e., $\eta^{ac} - \eta^{pa}$). For fair comparison we used the same task-sets for both modes. The x-axis of Fig. 4.2 represents the total system utilization (e.g., utilization of both real-time and security tasks). The positive values in the y-axis of Fig. 4.2 imply that the ACTIVE mode tasks obtain better tightness than the PASSIVE mode tasks.

The figure shows that ACTIVE mode tasks can achieve better cumulative tightness, and that the cumulative tightness η^{pa} is comparatively better in low to medium utilization. The main reason is that in ACTIVE mode security tasks are allowed to execute with higher priority, that causes less interference and eventually increases the feasible region in the optimization problems (and hence provides better tightness). For higher utilizations the difference is close to zero. This is because, as utilization increases there is less slack in the system, making it difficult to schedule security tasks frequently and resulting in similar levels of tightness for both modes.

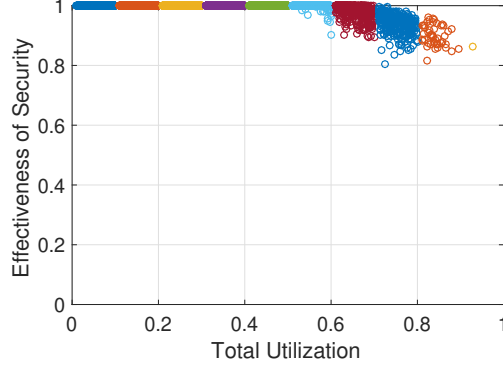


Figure 4.3: The effectiveness of security vs. total utilization of the system. The closer the y-axis values to 1, the nearer each security task’s period is to the desired period.

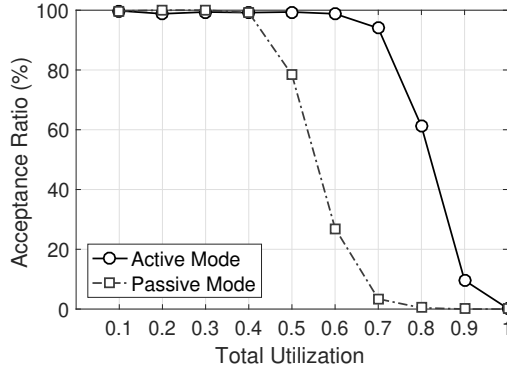


Figure 4.4: Schedulability of real-time and security tasks in both modes. The acceptance ratio is defined by the ratio of the number of accepted task sets over the total number of generated tasks. For each of the data points, 500 individual task-sets were tested.

Effectiveness of Security. The parameter $\eta^{(\cdot)}$ is given by the total number of security tasks and provides insights on cumulative measures of security. However, in this experiment (refer to Fig. 4.3) I wanted to measure the effectiveness of the security of the system by observing whether *each* of the security tasks in any mode can achieve an execution frequency closer to the desired one. Hence I used the following metric: $\xi = 1 - \frac{\|\mathbf{T}^* - \mathbf{T}^{\text{des}}\|_2}{\|\mathbf{T}^{\text{max}} - \mathbf{T}^{\text{des}}\|_2}$ where \mathbf{T}^* is the solution obtained from Algorithm 4.1, $\mathbf{T}^{\text{des}} = [T_i^{\text{des}}]_{\forall \tau_i}^T$ and $\mathbf{T}^{\text{max}} = [T_i^{\text{max}}]_{\forall \tau_i}^T$ are the desired and maximum period vector (refer to Section 4.7.1), respectively, and $\|\cdot\|_2$ denotes the Euclidean norm. The closer the value of ξ to 1, the nearer each of the security task’s period is to the desired period.

As the total utilization increases, the feasible set of the period adaptation problem that respects all constraints in the optimization problems becomes more restrictive. As a result, we see the degradation in effectiveness (in terms of ξ) for the task-sets with higher utilization. However, from my experiments I find that CONTEGO can achieve periods that are *within 18% of the desired periods*.

Impact on the Schedulability. I used the *acceptance ratio* metric to evaluate schedulability. The acceptance ratio (y-axis in Fig. 4.4) is defined as the number of accepted task-sets (e.g., the task-sets that satisfied all the constraints) over the total number of generated ones. As depicted in Fig. 4.4 the ACTIVE mode task-set achieves better schedulability compared to the PASSIVE ones. Recall that ACTIVE mode task-sets can be promoted up to priority level l_S . As a result ACTIVE mode security tasks potentially experience less interference than the PASSIVE ones. This flexibility gives the optimization routines a larger feasibility region to satisfy all the constraints.

4.7.2 Experiment with Security Applications in an Embedded Platform

To observe the performance of the proposed scheme in a practical setup, I implemented CONTEGO on an embedded platform. My experimental platform [91] was configured with 1 GHz ARM Cortex-A8 single-core processor and 512 MB RAM. I used Linux as the operating system – that allowed me to utilize the existing Linux-based IDSEs (refer to Table 4.3) for the evaluation. Since the vanilla Linux kernel is unsuitable for hard real-time scheduling, I enabled the real-time capabilities with the Xenomai [92] 2.6.3 real-time patch (kernel version 3.8.13-r72) on top of an embedded Debian Linux console image.

I measured the WCET of the real-time and security tasks using ARM cycle counter registers (e.g., CCNT), giving us nanosecond-level precision. Since these registers are not enabled by default, I developed a Linux kernel module to access the registers from application codes. My prototype implementation was developed in C and uses a fixed-priority scheduler powered by the Xenomai real-time patch. Real-time and security tasks in the system were defined by Xenomai `rt_task_create()` function and were suspended after the completion of corresponding instances using the `rt_task_wait_period()` function.

Real-time Tasks. For a real-time application, I considered a UAV control system (refer to Table 4.2). I implemented it using an open-source UAV model [93]. The original application codes were based on the STM32F4 micro-controller (ARM Cortex M4) and developed for FreeRTOS [94]. Because of differences in library support and execution semantics, I updated the source codes accordingly and ported them to Linux.

Security Tasks. To integrate security in the aforesaid control system, I included additional security tasks. For the security tasks, I considered two lightweight open-source intrusion detection mechanisms, (i) Tripwire [60], that detects integrity violations by storing clean system state during initialization and using it later to detect intrusions by comparing the current system state against the stored clean values, and (ii) Bro [69] that monitors anomalies in network traffic. As Table 4.3 shows, I consider several security tasks in both modes, e.g., *protecting security task’s own binary files*, *protecting system binary and library files*, *monitoring network traffic*. In each mode, I set the desired and maximum allowable periods of the security tasks such that utilization of the security

Table 4.2: Real-Time Task Parameters for the UAV Control System

Task	Function	Period (ms)
Guidance	Select the reference trajectory (i.e., altitude and heading)	1000
Controller	Execute closed-loop control functions (e.g., actuator commands)	5000
Reconnaissance	Read radar/camera data, collect sensitive information and send data to the base control station	10000

Table 4.3: Security Tasks used in the Experiments

Task	Function	Mode
Check own binary of the security routine (Tripwire)	Scan files (viz., compare their hash value) in the following locations: <code>/usr/sbin/siggen</code> , <code>/usr/sbin/tripwire</code> , <code>/usr/sbin/twadmin</code> , <code>/usr/sbin/twprint</code> , <code>/usr/local/bro/bin</code>	ACTIVE
Check critical executables (Tripwire)	Scan file-system binary (<code>/bin</code> , <code>/sbin</code>)	ACTIVE and PASSIVE
Check critical libraries (Tripwire)	Scan file-system library (<code>/lib</code>)	ACTIVE
Monitor network traffic (Bro)	Scan predefined network interface (<code>en0</code>)	ACTIVE and PASSIVE

tasks did not exceed 50% of the total system utilization.

Experience and Evaluation

Performance Impact in Different Modes. In the first set of experiments, I measured the average CPU load when the security tasks were executing in PASSIVE and ACTIVE modes. For that, we executed the security tasks independently for 500 s in PASSIVE and ACTIVE modes and observed the CPU load using `/proc/stat` interface (represents the y-axis of Fig. 4.5). As Fig. 4.5 shows, running security tasks in ACTIVE mode increased the average CPU load compared to running them in PASSIVE mode. This is because ACTIVE mode contains more security tasks (e.g., 4 compared to 2, refer to Table 4.1) and they execute more frequently than in PASSIVE mode. Because of the nature of applications, most RTS prefer predictability over performance. The overhead of running security tasks in ACTIVE mode comes with increased security guarantees that will suffice for many RTS.

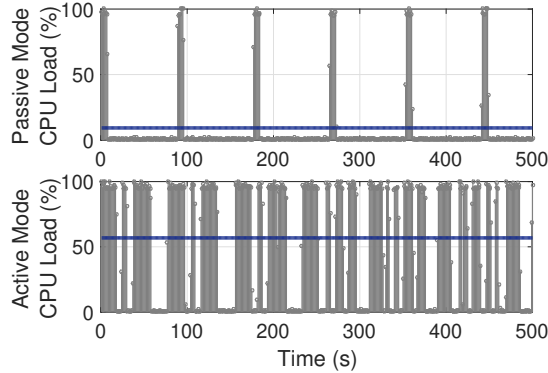


Figure 4.5: The CPU load when the security tasks executed in PASSIVE (top) and ACTIVE (bottom) mode, respectively. The horizontal line represents average load over the observation duration (500 s).

Impact on Detection Time. To study the detection performance I injected malicious code into the system that mimics anomalous behaviors. I assumed that an attacker can take over⁶ one of the low-priority real-time tasks (referred to as the victim task) and is able to insert malicious code that can execute with a privilege similar to that of legitimate tasks. I launched the attack at both the *network* and *host*-level. I defined network-level DoS attacks as too many rejected usernames and passwords submitted from a single address and used a real FTP DoS trace [95] to demonstrate the attack. Malware (such as LRK, tOrn, Adore) in general-purpose Linux environments causes damage to the system by modifying or overwriting the system binary [96, Ch. 5]. Thus I follow a similar approach to demonstrate a host-level attack, viz., I injected ARM shellcode [97] to override the victim task’s code and launched the attack by modifying the contents in the file-system binary. I obtained the periods of the security tasks in both modes by solving the period adaptation problem (Algorithm 4.1) and set it as the period of security tasks (by using the Xenomai `rt_task_set_periodic()` function). For each of the experiments, the workflow was as follows. I started with a clean (e.g., uncompromised) system state, launched the DoS attack at any random time of the program execution and then injected the shellcode after a random interval, and finally logged the time required by security tasks to detect the attacks. Initially the security tasks ran in PASSIVE mode. When the network-level attack was suspected by the security task (Bro), a mode change request was placed and the control was switched to ACTIVE mode with the corresponding ACTIVE mode tasks (see Table 4.3). As mentioned in Section 4.2, my focus is *not* on the effectiveness of a particular IDS here but on the effectiveness of integration of the IDSes into RTS. Therefore I controlled the experimental environment so that the results were not affected by the false positive/negative rates of the IDS used in the evaluation. In particular, both of the launched attacks were detectable by the respective IDSes used in the evaluation. Detection times were measured using ARM cycle counter registers (CCNT). To ensure the accuracy of the

⁶One way to override a task could be to use an approach similar to one presented in the literature [63] that exploits the deterministic behavior of the real-time scheduling.

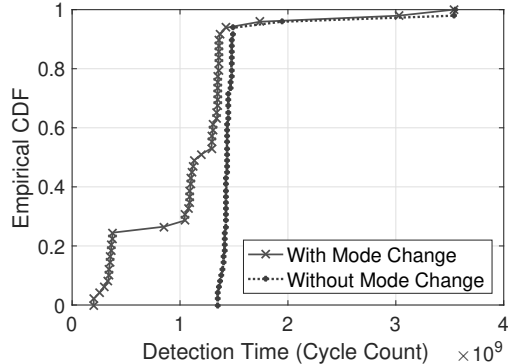


Figure 4.6: The empirical distribution of time to detect the intrusions when mode change was allowed vs when security tasks were run only in PASSIVE mode. I used ARM cycle counter registers to measure the detection time. A total of 50 individual experiment instances were examined to obtain the timing traces.

detection time measurements, I disabled all the frequency scaling features in the kernel (by using the `cpufrequtils` utility) and allowed the platform to execute with a constant frequency (i.e., 1 GHz, the maximum frequency of our experimental platform).

I compared the performance of CONTEGO with an alternative approach that has no provision for mode changes — that I refer to henceforth as the “opportunistic execution” scheme. In this scheme the security tasks are run with the lowest priority (similar to the PASSIVE mode of operation in CONTEGO). Specifically, I measured the time to detect both the host and network-level intrusions, and plot the empirical cumulative distribution function (CDF) of those detection times in Fig. 4.6. The x-axis in Fig. 4.6 represents the detection time (in cycle count) and the y-axis represents the probability that the attack would be detected by that time. The empirical CDF is defined as $\hat{F}_\alpha(j) = \frac{1}{\alpha} \sum_{i=1}^{\alpha} \mathbb{I}_{[\zeta_i \leq j]}$, where α is the total number of experimental observations, ζ_i is the time taken to detect the attack in the i -th experimental observation, and j represents the x -axis values (viz., the detection times in cycle count) in Fig. 4.6. The indicator function $\mathbb{I}_{[\cdot]}$ outputs 1 if the condition $[\cdot]$ is satisfied and 0 otherwise.

From Fig. 4.6 we can see that CONTEGO provides better detection time (i.e., fewer cycle counts required to detect the intrusions). From my experiments I find that *on average* CONTEGO detects attacks 27.29% faster than the reference scheme does. Opportunistic execution scheme allows the security tasks to run only when other real-time tasks are not running, leading to more interference (i.e., higher response times), and does not provide any mechanisms to adapt against abnormal behaviors (e.g., the DoS attack in the experiments). In contrast, CONTEGO allows quick response to anomalies (by switching to ACTIVE mode when a DoS attack is suspected). Since ACTIVE security tasks can run with higher priority and less interference without impacting the timeliness constraints of real-time tasks, CONTEGO had a superior detection rate in general for most of the experiments without impacting safety.

4.8 CONCLUSION

In this chapter I introduce a dual-mode framework, CONTEGO, to integrate security into legacy single core RTS and provide a glimpse of security design metrics for RTS. I demonstrate the efficacy of such integration mechanisms in a practical system and analyze the design trade-offs, both from security and real-time perspectives. The framework presented in this chapter asserts one part of my dissertation hypothesis (Section 1.1) since CONTEGO is a software-based technique that works at the scheduler-level and allows the designers to integrate periodic security checks while retaining real-time guarantees. I believe CONTEGO will provide valuable hints to the engineers on how to enhance security into such safety-critical systems.

I note that CONTEGO targets single core systems since majority of legacy RTS are built on single core chips. However, modern real-time CPS are migrating towards multicore platforms [23]. This makes the problem of integrating security mechanisms more complex. This is because, multicore platforms allow parallel execution of security tasks and designers have multiple choices for where to allocate the security tasks. Chapter 5 addresses the problem of integrating security tasks in a multicore setup.

CHAPTER 5: A DESIGN-SPACE EXPLORATION FOR INTEGRATING SECURITY TASKS IN MULTICORE REAL-TIME SYSTEMS

This chapter further asserts one part of my dissertation hypothesis (Section 1.1) and shows the feasibility of integrating security monitoring mechanisms into *multicore real-time platforms* by using time-aware scheduler-level techniques. Recall that the framework proposed in Chapter 4 (CONTEGO) targets single core systems. Despite the fact that most legacy real-time applications are designed using platforms equipped with a single-core CPU, the trend towards multicore systems can be seen as many off-the-shelf devices nowadays are built on top of a multicore environment [23]. As the use of multicore platforms in safety-critical RTS is increasingly becoming common, the focus of this work is on integrating or retrofitting security mechanisms into *multicore RTS*. This makes the problem of integrating security mechanisms more complex, as designers now have multiple choices for where to allocate the security tasks. In this chapter I propose two design-time frameworks, viz., HYDRA and HYDRA-C, that allows security tasks to operate with existing real-time tasks in multicore platforms *without* perturbing system parameters or normal execution patterns. HYDRA uses a static partitioning approach where security tasks are allocated to the cores (i.e., do not migrate across cores at runtime). HYDRA-C, in contrast, executes security tasks in a “continuous” manner — i.e., as often as possible, across cores. This is to ensure that any such mechanisms run with few interruptions, if any. I evaluate HYDRA and HYDRA-C using a proof-of-concept implementation with intrusion detection mechanisms as security tasks. I develop and use both, (a) a custom intrusion detection system (IDS) as well as (b) Tripwire — an open source data integrity checking tool. I find that my methods do not impact the schedulability and, on average, HYDRA-C can detect intrusions 19.05% faster when compared to HYDRA without impacting the performance of real-time tasks.

5.1 INTRODUCTION

Multicore processors have found increased use in the design of modern RTS [23]. However, the use of such processors increases the security problems (e.g., due to parallel execution of critical tasks) [37]. In this chapter I *evaluate design alternatives to improve the security posture of multicore RTS* through integration of “security tasks” while ensuring that the existing real-time tasks are not affected by such integration. My focus here is to integrate security in an existing (e.g., legacy) system where it is harder to (a) modify the micro-architecture (say inclusion of extra hardware/processor cores) or (b) change real-time task parameters (such as execution time and/or period). Existing work that integrate security in RTS either focuses on single core systems (e.g., see Chapter 4 and related work [10, 11, 42, 58, 59]) and/or require modification of system parameters [10, 11, 13, 42, 58, 59] and thus are not applicable for systems where it is harder to change the real-time task parameters. My main goal is to *explore design mechanisms that can raise the responsiveness of such monitoring tasks by increasing their frequency of execution*. Unlike single

core systems, integrating security into multicore platforms is more challenging since designers have multiple choices across cores (due to parallel execution of tasks) to retrofit security mechanisms. For instance, one design choice could be *statically assign cores for security tasks* (in conjunction with the real-time tasks). The challenge then is to determine *core allocation* and *periods* of the security tasks. Another possibility is to *execute them continuously across any available core* to provide better security monitoring and detection. As an example, consider an intrusion detection system (IDS) e.g., that checks the integrity of file systems. If such a system is interrupted (before it can complete entire checking), then an adversary could use that opportunity to intrude into the system and, perhaps, stay resident in the part of the filesystem that has already been checked. If, in contrast, the IDS task is able to execute with as few interruptions as possible (e.g., by moving immediately to an empty core when it is interrupted), then there is much higher chance of successful detection and correspondingly, a much lower chance of successful adversarial action.

In this chapter I present two design-time frameworks, HYDRA¹ and HYDRA-C², for partitioned³ RTS. I first start with static security integration policy (i.e., does *not* allow runtime migration), HYDRA, and find a suitable *core assignment* of security tasks in order to ensure that they can *execute with a frequency close to what a designer expects* (Section 5.3). As we shall see in Section 5.5, this static partitioning of security tasks results in delayed detection of intrusions. I then extend HYDRA with an alternate design method, HYDRA-C, that uses the concept of semi-partitioned scheduling [31] to enable *continuous execution* of security tasks (i.e., execute as frequently as possible) across cores without impacting the timing constraints of other, existing, real-time tasks (Section 5.4).

Contributions. In this chapter I present the following contributions.

- Integrating security mechanisms in a multicore setup where changing existing real-time task parameters is not an option.
- A mathematical model to jointly obtain the assignment of security tasks to respective cores with execution frequency close to the desired values (Section 5.3).
- A mathematical model and iterative solution that allows security tasks to execute as frequently as possible while still considering the schedulability constraints of other tasks (Section 5.4).

I also present an implementation on a realistic ARM-based multicore rover platform (Section 5.5.1) and carry out a design-space exploration to study the trade-offs for schedulability and security (Section 5.5.2). My evaluation shows that HYDRA-C can achieve better execution

¹In Greek mythology Hydra is a serpent with multiple heads. I refer to my scheme as HYDRA since I am trying to maximize the potential across multiple ‘heads’ (cores).

²HYDRA-C stands for “HYDRA-Continuous”.

³Since this is the commonly used multicore scheduling approach for many commercial and open-source OSs (such as OKL4 [98], QNX [99], real-time Linux [100], etc.) — mainly due to its simplicity and efficiency [101].

frequency (consequently quicker intrusion detection) when compared with both fully-partitioned (i.e., HYDRA) and global scheduling approaches while providing same or better schedulability.

Note: I do not target my frameworks towards any specific security mechanism — my focus is to integrate any designer-provided security technique into a multicore-based RTS. I used *Tripwire* and my *in-house custom-developed malicious kernel module checker* to demonstrate the feasibility of my approach (Section 5.5) — the solutions proposed in this chapter is more broadly applicable to other security mechanisms.

5.2 MODEL AND ASSUMPTIONS

5.2.1 Real-time Tasks and Scheduling Model

Consider a set of N_R real-time tasks $\Gamma_R = \{\tau_1, \tau_2, \dots, \tau_{N_R}\}$, scheduled on a multicore platform with M identical cores $\mathcal{M} = \{\pi_1, \pi_2, \dots, \pi_M\}$. Each real-time task τ_r is represented by the tuple (C_r, T_r, D_r) where C_r is the worst-case execution time (WCET), T_r is the minimum inter-arrival time (i.e., period) and D_r is the relative deadline. In this work, I consider *partitioned fixed-priority preemptive scheduling* [23] since (a) it does not introduce task migration costs and (b) it is widely supported in many commercial and open-source real-time OSs (e.g., QNX, OKL4, real-time Linux, etc.). I assume constrained deadlines for real-time tasks (i.e., $D_r \leq T_r$) and the task priorities are assigned according to rate-monotonic (RM) [80] order. All events in the system happen with the precision of integer clock ticks. Real-time tasks are scheduled using partitioned fixed-priority preemptive scheme [23, 101]. I further assume that the real-time tasks are *schedulable*, viz., the worst-case response time (WCRT), denoted as \mathcal{R}_r , is less than deadline. Therefore, the following necessary and sufficient schedulability condition holds for each real-time tasks τ_r assigned to any given core π_m [101]:

$$\exists t : 0 < t \leq D_r \text{ and } C_r + \sum_{\tau_i \in hp(\tau_r, \pi_m)} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t, \quad (5.1)$$

where $hp(\tau_r, \pi_m)$ denotes the set of real-time tasks with higher priority than τ_r assigned to core π_m .

5.2.2 Security Task Integration

I now formally define security tasks. I propose to improve the security posture by integrating additional N_S periodic security tasks $\Gamma_S = \{\tau_1, \tau_2, \dots, \tau_{N_S}\}$ (i.e., tasks that are specifically designed for monitoring purposes). I follow the sporadic security task model introduced earlier (Section 4.3.2). In Section 5.3, I first characterize each security task τ_s by the tuple $(C_s, T_s^{des}, T_s^{max})$ where C_s is the WCET, T_s^{des} is the best period (minimum inter-arrival time) between successive releases (i.e., $F_s^{des} = \frac{1}{T_s^{des}}$ is the desired frequency for effective security monitoring and/or intrusion detection) and T_s^{max} is a designer provided upper bound of the period — if the period of the

security task is larger than T_s^{max} then the responsiveness is too low and security checking may not be effective. Section 5.4 further relaxes the security task model and represents each security task as (C_s, T_s, T_s^{max}) to allow it execute as frequently as possible, across cores. I assume that the priorities of the security tasks are distinct and specified by the designers (e.g., derived from specific security requirements). These tasks have implicit deadlines, i.e., they need to finish execution before the next invocation. I also assume that task migration and context switch overhead is negligible compared to the WCETs.

I first propose a mechanism, HYDRA, statically allocate security tasks to their respective cores (Section 5.3). I then extend HYDRA with an alternative design choice (named HYDRA-C) that allows runtime migration of security tasks (Section 5.4). As we shall see in Section 5.5, HYDRA-C provides better security (i.e., faster detection time) and schedulability but comes with a cost (increases context switch overhead).

5.3 HYDRA: FIXED ASSIGNMENT OF SECURITY TASKS

Recall that one way to integrate security mechanisms into existing systems without perturbing real-time task behavior is to execute security tasks with the *lowest priority* as compared to the real-time tasks (Chapter 4). Thus security tasks will execute *opportunistically* in the slack time (e.g., when other real-time tasks are not running). As mentioned in Section 4.1, if the interval between consecutive monitoring events is too large, the adversary may remain undetected and harm the system between two invocations of the security task. In contrast, very frequent execution of security tasks may impact the schedulability of the system (due to higher utilization). Since the actual periods of the security tasks are not known and we need to *adapt* the periods to optimize the trade-offs between schedulability and defense against intrusions.

I measure the security of the system by means of the *achievable periodic monitoring* and our goal is to minimize the perturbation between the achievable (unknown) period T_s and the given desired period T_s^{des} for all security tasks $\tau_s \in \Gamma_S$. Therefore I consider the following “tightness” metric introduced in Section 4.4:

$$\eta_s = \frac{T_s^{des}}{T_s}, \quad (5.2)$$

that represents how close the period of the security task is to its desired period. Note that the tightness metric is bounded by $\frac{T_s^{des}}{T_s^{max}} \leq \eta_s \leq 1$. As mentioned earlier, if the interval between consecutive monitoring events is too large, the adversary may remain undetected and harm the system between two invocations of the security task. In contrast, very frequent execution of security tasks may impact the schedulability of the system (due to higher utilization). The metric in Eq. (5.2) allows me to measure how close the security tasks are able to get to their desired monitoring frequencies.

One fundamental problem while integrating security mechanisms is to determine *which security tasks will be assigned to which core and executed when*. Although security tasks can execute in any

of the M available cores and any period $T_s^{des} \leq T_s \leq T_s^{max}$ is acceptable, the actual task-to-core assignment and the periods of the security tasks are not known a priori. The goal of HYDRA therefore is to jointly find the core-to-task assignment and suitable periods for security tasks. Note that arbitrarily setting $T_s = T_s^{des}$ for all (or some) security tasks $\tau_s \in \Gamma_S$ may lead to the system becoming *unschedulable*. This is because, low-priority security tasks may miss deadlines due to interference from higher priority tasks. Also exhaustively finding all possible acceptable periods for the security tasks for all available cores is not feasible. It will cause an exponential blow-up as numbers of tasks and cores increase. For instance for a given taskset Γ_S , there is a total of $|\mathcal{M} \times \Gamma_S|$ assignments possible⁴ (where $A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$ and $|\cdot|$ denotes set cardinality) and for each combination the period for each security task $\tau_s \in \Gamma_S$ can be any value within the range $[T_s^{des}, T_s^{max}]$. In order to address this combinatorial problem I obtain the periods of the security tasks by framing it as an optimization problem.

5.3.1 Formulation as an Optimization Problem

Objective Function and Bounds on Period

Let me consider the vector $\mathbf{X} = [x_s^m]_{\forall \tau_s \in \Gamma_S, \forall \pi_m \in \mathcal{M}}^T$ where $x_s^m = 1$ if τ_s is assigned to π_m and 0 otherwise. Recall that my goal is to find a task assignment that minimizes the difference between achievable and desired periods (e.g., maximize the tightness) for all the security tasks. Hence I define the following objective function:

$$\max_{\mathbf{X}, \mathbf{T}} \sum_{\pi_m \in \mathcal{M}} \sum_{\tau_s \in \Gamma_S} x_s^m \omega_s \eta_s = \sum_{\pi_m \in \mathcal{M}} \sum_{\tau_s \in \Gamma_S} x_s^m \omega_s \frac{T_s^{des}}{T_s} \quad (5.3)$$

where $\mathbf{T} = [T_s]_{\forall \tau_s \in \Gamma_S}^T$ is the (unknown) period vector that needs to be determined and ω_s reflects the priority (higher priority tasks would have large ω_s). Besides, in order to satisfy the frequency of periodic monitoring, the security task needs to satisfy the following constraint:

$$T_s^{des} \leq T_s \leq T_s^{max}, \quad \forall \tau_s \in \Gamma_S. \quad (5.4)$$

Finally, each security task must be assigned to exactly one core: $\sum_{\pi_m \in \mathcal{M}} x_s^m = 1, \quad \forall \tau_s \in \Gamma_S$.

Schedulability Constraint

Since the security tasks are executed with a priority lower than all real-time tasks, they will suffer interference from all real-time and high priority security tasks executing in the same core. Let $hp_S(\tau_s) \subset \Gamma_S$ denote the set of security tasks with a higher priority than τ_s . The worst-case

⁴For instance, when $M = 8$ cores and $N_S = 10$ tasks there is a total of $3.518437208883 \times 10^{13}$ possible assignments.

release pattern of τ_s occurs when τ_s and all high-priority tasks are released simultaneously [81]. Using response time analysis [84] I calculate an upper bound to the interference experienced by τ_s for a given core π_m and represent as follows:

$$I_s^m = \sum_{\tau_r \in \Gamma_R} \mathbb{I}_r^m \left(1 + \frac{T_s}{T_r}\right) C_r + \sum_{\tau_h \in hp_S(\tau_s)} x_h^m \left(1 + \frac{T_s}{T_h}\right) C_h, \quad (5.5)$$

where $\mathbb{I}_r^m = 1$ if the real-time task τ_r is partitioned to core π_m and 0 otherwise.

The first and second term in Eq. (5.5) represent the amount of interference from real-time and high-priority security tasks, respectively. Note that the assignment of real-time tasks to cores is known by assumption. In order to ensure that each security task τ_s will complete its execution before its deadline on its assigned core, the following constraint needs to be satisfied:

$$C_s + I_s^m \leq T_s, \quad \forall \tau_s \in \Gamma_s, \forall \pi_m \in \mathcal{M} : x_s^m = 1. \quad (5.6)$$

The variables \mathbf{X} and \mathbf{T} in the above formulation turn the problem into a *non-linear combinatorial optimization problem* that is NP-hard. I therefore propose an iterative algorithm HYDRA that jointly finds the security tasks' period and core assignment.⁵

5.3.2 Algorithm

As mentioned earlier, jointly finding the security task assignment and periods is an NP-hard problem. Even for fixed periods, finding the assignment for security tasks turns the problem to a bin-packing problem that is known to be NP-hard [102]. Existing partitioning heuristics (e.g., first-fit, best-fit, etc.) [23] are *not* directly applicable in our context since the real-time requirements (e.g., minimize the number of cores so that all real-time tasks can meet deadlines) are often different from the security requirements (e.g., execute security tasks more often to improve intrusion detection rate without violating real-time constraints).

For a given task τ_s and allocation vector \mathbf{X} , let us rewrite the optimization problem as follows:

$$\max_{T_s} \eta_s, \text{ Subject to: } T_s^{des} \leq T_s \leq T_s^{max}, C_s + I_s^m \leq T_s. \quad (5.7)$$

Notice that for a given assignment \mathbf{X} (see Algorithm 5.1), the period T_s is the only variable (when the $T_h, \forall \tau_h \in hp_S(\tau_s)$ is known) in I_s^m (see Eq. (5.5)). Although the period adaptation problem in Eq. (5.7) is a constraint optimization problem it can be transformed into a convex optimization problem (that is solvable in polynomial time). For details of this reformulation I refer the readers

⁵Appendix B.2 presents a comparison of my proposed iterative scheme with a brute force approach that exhaustively searched for all possible combinations for a small setup ($M = 2$ cores and up to $N_S = 6$ security tasks). My experiments show that the performance degradation (in cumulative tightness) is less than 22% and that may be acceptable given the exponential computational complexity of finding an optimal solution using exhaustive search.

Algorithm 5.1: HYDRA — Task Allocation and Period Adaptation

Input: Input taskset $\Gamma = \{\Gamma_R \cup \Gamma_S\}$ and the partition of real-time tasks $\mathbb{I} = [\mathbb{I}_r^m]^T_{\forall \tau_r \in \Gamma_R, \forall \pi_m \in \mathcal{M}}$
Output: The security task allocation $\mathbf{X} = [x_s^m]^T_{\forall \tau_s \in \Gamma_S, \forall \pi_m \in \mathcal{M}}$ and periods $\mathbf{T} = [T_s]^T_{\forall \tau_s \in \Gamma_S}$, if the taskset is schedulable, **Unschedulable** otherwise.

- 1: Initialize $x_s^m := 0, \forall \tau_s \in \Gamma_S, \forall \pi_m \in \mathcal{M}$
- 2: **for each** security task $\tau_s \in \Gamma_S$ (from higher to lower priority) **do**
- 3: **for each** core $\pi_m \in \mathcal{M}$ **do**
- 4: Solve the optimization problem in Eq. (5.7)
- 5: **end for**
- 6: Let $\mathcal{M}'_s \subseteq \mathcal{M}$ is the set of core(s) for which the optimization problem is feasible
- 7: **if** $\mathcal{M}'_s = \emptyset$ **then**
- 8: */* Unable to find suitable period for τ_s */*
- 9: **return** **Unschedulable**
- 10: **end if**
- 11: Find the core $\pi_{m^*} = \operatorname{argmax}_{\pi_m \in \mathcal{M}'_s} \eta_s^m$ where η_s^m is the tightness of τ_s obtained for π_m
- 12: Set $x_s^{m^*} := 1$ */* Assign τ_s to π_{m^*} */*
- 13: Update period $T_s := T_s^{m^*}$ where $T_s^{m^*}$ is the period obtained by solving optimization for π_{m^*}
- 14: **end for**
- 15: **return** (\mathbf{X}, \mathbf{T}) */* Return the allocation vector and periods */*

to Appendix B.1.

The proposed HYDRA algorithm (summarized in Algorithm 5.1) works as follows. I start with the highest priority security task τ_s and try to obtain the best period for each available core $\pi_m \in \mathcal{M}$ by solving the period adaptation problem introduced in Eq. (5.7) (Line 4). If there exists a set of cores $\mathcal{M}'_s \subseteq \mathcal{M}$ for which the optimization problem is feasible (e.g., an optimal period is obtained satisfying the real-time constraints) we pick the core $\pi_{m^*} \in \mathcal{M}'_s$ that gives the maximum tightness (Line 11) and allocate the security task to core π_{m^*} (Line 12). This will ensure that the more critical security tasks will get a period close to the desired one. I repeat this process for all security tasks to jointly obtain the assignment and periods. If for any security task τ_j the set of available cores \mathcal{M}'_j is empty (e.g., the optimization problem is infeasible) we return the taskset as *unschedulable* (Line 9) since it is not possible to find any suitable core with given taskset parameters. This unschedulability result will provide hints to the designers to update the parameters of security tasks (and/or the real-time tasks, if possible) in order to integrate security for the target system.

5.4 HYDRA-C: CONTINUOUS SECURITY MONITORING

Recall from Section 5.1 that my goal is to explore the possible ways in which security could be integrated in multicore-based real-time platforms. The HYDRA mechanism presented in Section 5.3 assumes that the real time tasks are distributed across *all* available cores. I now propose an alternative design choice where I allow security tasks to continuously migrate at runtime (i.e., the combined taskset with real-time and security tasks follows a semi-partitioned scheduling policy) whenever any core is available (e.g., when other real-time or higher-priority security tasks are not

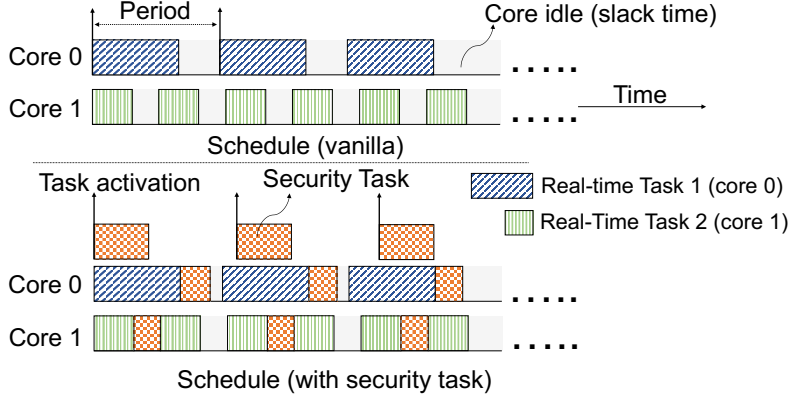


Figure 5.1: Illustration of HYDRA-C for a dual-core platform: two real-time tasks (blue and green) are statically assigned to two cores (core 0 and core 1, respectively). I propose to integrate a security task (red) that will execute with lowest priority and can be migrated to either core (whichever is idle) at runtime.

running). I refer to this scheme as HYDRA-C. An illustration of HYDRA-C is presented in Fig. 5.1 where two real-time tasks (represented by blue and green rectangles) are partitioned into two cores and a newly added security task (red rectangle) can move across cores. As we shall see in Section 5.5, allowing security tasks to execute on any available core will give us the opportunity to execute security tasks more frequently (e.g., with shorter period) and that leads to better responsiveness (faster intrusion detection time).

5.4.1 Period Selection

As mentioned earlier, one fundamental question is to figure out *how often to execute security tasks* so that the system remains schedulable and also can execute within a designer provided frequency bound (so that the security checking remains effective). Mathematically period selection can be expressed as: minimize $\sum_{T_s, \forall \tau_s \in \Gamma_S} T_s$, subject to $\mathcal{R}_s \leq T_s \leq T_s^{max}, \forall \tau_s \in \Gamma_S$. This is a non-trivial optimization problem since the period of τ_s can be anything in $[\mathcal{R}_s, T_s^{max}]$ and the response time \mathcal{R}_s is a variable as it depends on the period of other higher priority security tasks. I first derive the WCRT of the security tasks and use it as a (lower) bound to find the periods (Section 5.4.2).

Response Time Analysis

In the following I determine the response time of a job τ_s^k of security task τ_s using an iterative method and the response time in each iteration is denoted by x .

Interference Caused by Real-Time Tasks. The interference $I_{\tau_s \leftarrow \tau_i}$ caused by a task τ_i on τ_s^k is the number of time units in the busy period⁶ when τ_i executes while τ_s^k does not. I first

⁶This is the maximal continuous time interval $[t_1, t_2)$ until τ_s^k finishes where all the cores are executing either

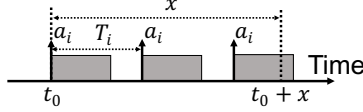


Figure 5.2: Workload of the real-time tasks for a window of size x . a_i denotes the arrival time.

calculate the *workload*⁷ of the real-time tasks using the following lemma and use this to derive the interference.

Lemma 5.1. *The maximum workload of real-time tasks executed on a given core π_m (in any possible time interval of length x) is obtained when all real-time tasks are released synchronously at the beginning of the interval.*

Since real-time tasks are statically partitioned to cores and they have higher priority than any task that is allowed to migrate between cores, their worst-case workload can be obtained based on the critical instant [80] used for single-core fixed-priority scheduling case (formal proof in Appendix B.3).

Let $\Gamma_R^{\pi_m} \subseteq \Gamma_R$ denote the set of real-time tasks partitioned to core π_m . Based on Lemma 5.1, an upper bound to the workload of real-time tasks on π_m can be obtained by assuming that each real-time task τ_r is released at the beginning of the interval and each job of τ_r executes as early as possible after being released (see Fig. 5.2). I thus obtain the workload for real-time task τ_r : $W_r^R(x) = \left\lfloor \frac{x}{T_r} \right\rfloor C_r + \min(x \bmod T_r, C_r)$ and summing over all real-time tasks on π_m yields a total workload $\sum_{\tau_i \in \Gamma_R^{\pi_m}} W_i^R(x)$. Note that by definition, the interference caused by a group of tasks executing on the same core π_m on τ_s cannot be greater than $x - C_s + 1$. Therefore, the maximum interference caused by real-time tasks can be bounded as:

$$I_{\tau_s \leftarrow \Gamma_R^{\pi_m}}(x, \sum_{\tau_i \in \Gamma_R^{\pi_m}} W_i^R(x)) = \min \left(\sum_{\tau_i \in \Gamma_R^{\pi_m}} W_i^R(x), x - C_s + 1 \right). \quad (5.8)$$

Interference Caused by Other Security Tasks. I next consider the workload of security tasks with higher priority than τ_s . The workload computation for this case depends on the arrival time of the task relative to the beginning of the busy period. Let me define a task τ_i as a *carry-in* task (CI) if there exists one job of τ_i that has been released before the beginning of a given time window of length x and executes within the window. If no such job exists, τ_i is referred to as a *non-carry-in* task (NC).

To calculate the number of carry-in tasks, I extend the busy period of τ_s^k from its arrival time (denoted by a_s) to an earlier time instance t_0 (see Fig. 5.3) such that during any time instance $t \in [t_0, a_s)$ all cores are busy executing tasks with higher priority than τ_s [103]. Note that by

higher priority tasks or τ_s^k itself.

⁷The workload $W_i(w)$ of a task τ_i in a window of length w represents the accumulated execution time of τ_i within this time interval [103].

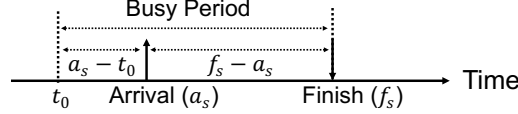


Figure 5.3: Busy period extension.

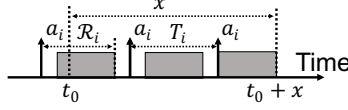


Figure 5.4: Illustration of carry-in task for a window of size x .

definition, this implies that there was at least one free core (i.e., not executing higher priority tasks) at time $t_0 - 1$.

Lemma 5.2. *At most $M - 1$ higher priority tasks can have carry-in at time t_0 .*

Proof. The maximum number of higher priority tasks that can have carry-in at t_0 is $M - 1$ since by definition there have to be strictly less than M higher priority tasks active at time $t_0 - 1$ (otherwise they will occupy all the cores). QED.

Since Lemma 5.2 holds for all tasks with higher priority than τ_s , an immediate corollary is that the number of security tasks with carry-in at t_0 also cannot be larger than $M - 1$. If a security task τ_i does not have carry-in, its workload is maximized when the task is released at the beginning of the busy interval. Hence, we can calculate the workload bound $W_i^{SNC}(x)$ for the interval x as follows: $W_i^{SNC}(x) = \left\lfloor \frac{x}{T_i} \right\rfloor C_i + \min(x \bmod T_i, C_i)$. Likewise, the workload bound for a carry-in security task τ_i in an interval of length x starting at t_0 is given by (see Fig. 5.4): $W_i^{SCI}(x) = W_i^{SNC}(\max(x - \bar{x}_i, 0)) + \min(x, C_i - 1)$, where $\bar{x}_i = C_i - 1 + T_i - \mathcal{R}_i$. We can bound the workload of the first carry-in job to $C_i - 1$ because the job must have started executing at the latest at $t_0 - 1$ (given that not all cores are busy). Finally, using the same argument as before, the interference of τ_i can be bounded as follows: $I_{\tau_s \leftarrow \tau_i}(x, W_i(x)) = \min(W_i(x), x - C_s + 1)$, where $W_i(x)$ is either $W_i^{SNC}(x)$ or $W_i^{SCI}(x)$. Notice that the WCRT and periods of security task in the carry-in workload function is actually an unknown parameter. However, I follow an iterative scheme (Section 5.4.2) that allows me to calculate the period and WCRT of all higher priority security tasks before I calculate the interference for task τ_s .

Response Time Calculation

Let $hp_S(\tau_s)$ denote the set of security tasks with a higher priority than τ_s . Note that we do not know which (at most) $M - 1$ security tasks in $hp_S(\tau_s)$ have carry-in. In order to derive the WCRT of τ_s , let us define $\mathcal{Z}_{\tau_s} \subset \Gamma \times \Gamma$ as the set of all partitions of $hp_S(\tau_s)$ into two subsets Γ_s^{NC} and Γ_s^{CI}

(i.e., the non overlapping set of carry-in and non-carry-in tasks) such that:

$$\Gamma_s^{NC} \cap \Gamma_s^{CI} = \emptyset, \Gamma_s^{NC} \cup \Gamma_s^{CI} = hp_S(\tau_s), \text{ and } |\Gamma_s^{CI}| \leq M - 1. \quad (5.9)$$

For a given carry-in and non-carry-in set (i.e., Γ_s^{NC} and Γ_s^{CI}), the total interference experienced by τ_s is calculated as follows:

$$\begin{aligned} \Omega_s(x, \Gamma_s^{NC}, \Gamma_s^{CI}) = & \sum_{\pi_m \in \mathcal{M}} I_{\tau_s \leftarrow \Gamma_R^{\pi_m}}(x, \sum_{\tau_i \in \Gamma_R^{\pi_m}} W_i^R(x)) + \\ & \sum_{\tau_i \in \Gamma_s^{NC}} I_{\tau_s \leftarrow \tau_i}(x, W_i^{SNC}(x)) + \sum_{\tau_i \in \Gamma_s^{CI}} I_{\tau_s \leftarrow \tau_i}(x, W_i^{SCI}(x)). \end{aligned} \quad (5.10)$$

The response time $\mathcal{R}_{s|(\Gamma_s^{NC}, \Gamma_s^{CI})}$ then will be the minimal solution of the following iteration⁸ [103]: $x = \left\lfloor \frac{\Omega_s(x, \Gamma_s^{NC}, \Gamma_s^{CI})}{M} \right\rfloor + C_s$. I solve this using an iterative fixed-point search with the initial condition $x^{(0)} = C_s$. The search terminates if there exists a solution (i.e., $x = x^{(l)} = x^{(l-1)}$ for some iteration l) or when $x^{(l)} > T_s^{max}$ for any iteration l since τ_s becomes trivially unschedulable for WCRT greater than T_s^{max} . Finally, I calculate the WCRT of τ_s as follows:

$$\mathcal{R}_s = \max_{(\Gamma_s^{NC}, \Gamma_s^{CI}) \in \mathcal{Z}_{\tau_s}} \mathcal{R}_{s|(\Gamma_s^{NC}, \Gamma_s^{CI})}. \quad (5.11)$$

5.4.2 Algorithm

The security task τ_s remains schedulable with any period $T_s \in [\mathcal{R}_s, T_s^{max}]$. However as mentioned earlier, the calculation of \mathcal{R}_s requires us to know the period and response time of other high priority tasks $\tau_h \in hp_S(\tau_s)$. Also if we arbitrarily set $T_s = \mathcal{R}_s$ (since this allows us to execute security tasks more frequently) it may negatively affect the schedulability of other tasks that are at a lower priority than τ_s because of a high degree of interference from τ_s . Hence, I developed an iterative algorithm that trades-off between schedulability and monitoring frequency.

My proposed solution (Algorithm 5.2) works as follows. I first fix the period of each security task T_s^{max} and calculate the response time \mathcal{R}_s (Line 1). If there exists a task τ_j such that $\mathcal{R}_j > T_j^{max}$ I report the taskset as unschedulable (Line 3) since it is not possible to find a period for the security tasks within the designer provided bounds — this unschedulability result will help the designer in modifying the requirements (and perhaps real-time tasks' parameters, if possible) accordingly to integrate monitoring tasks for the target system. If the taskset is schedulable with T_s^{max} , I then optimize the periods from higher to lower priority order (Lines 5–8) and return the period (Line 9). To be specific, for each task $\tau_s \in \Gamma_S$ I perform a logarithmic search [104, Ch. 6] (see Algorithm 5.3 for the pseudocode) and find the minimum period T_s^* within the range $[R_s, T_s^{max}]$ such that all low priority tasks (denoted as $lp(\tau_s)$) remain schedulable, i.e., $\forall \tau_j \in lp(\tau_s) : \mathcal{R}_j \leq T_j^{max}$ (Line 7)

⁸Note that the worst-case is when the job arrives at t_0 (i.e., $a_s = t_0$).

Algorithm 5.2: HYDRA-C — Period Selection

Input: Set of real-time and security tasks $\Gamma = \Gamma_R \cup \Gamma_S$ **Output:** Periods of the security tasks, \mathbf{T} (if the security tasks are schedulable); **Unschedulable** otherwise

```
1: Set  $T_s := T_s^{max}$  and calculate  $\mathcal{R}_s$  for  $\forall \tau_s \in \Gamma_S$ 
2: if  $\exists \tau_s$  such that  $\mathcal{R}_s > T_s^{max}$  then
3:   return Unschedulable
4: end if
5: for each security task  $\tau_s \in \Gamma_S$  (from higher to lower priority) do
6:   /* Find period for which all lower priority tasks are schedulable */
7:   Find minimum  $T_s^* \in [\mathcal{R}_s, T_s^{max}]$  using Algorithm 5.3 such that all low priority task  $\tau_j$  remain schedulable
   (i.e.,  $\mathcal{R}_j \leq T_j^{max}, \forall \tau_j$ )
8: end for
9: return  $\mathbf{T} := [T_s^*]_{\forall \tau_s \in \Gamma_S}$  /* return the periods */
```

Algorithm 5.3: Calculation of Minimum Feasible Period for τ_s

Input: Set of real-time and security tasks $\Gamma = \Gamma_R \cup \Gamma_S$ **Output:** A feasible period T_s^* for the security task under analysis (i.e., τ_s)

```
1: Define  $T_s^l := \mathcal{R}_s$ ,  $T_s^r := T_s^{max}$ ,  $T_s^c := 0$ 
2: Set  $\widehat{T}_s := \{T_s^{max}\}$  /* Initialize to store the set of feasible periods */
3: while  $T_s^l <= T_s^r$  do
4:   Update  $T_s^c := \lfloor \frac{T_s^l + T_s^r}{2} \rfloor$ 
5:   if  $\exists \tau_j \in lp(\tau_s)$  such that  $\tau_j$  is not schedulable with  $T_s = T_s^c$  then
6:     /* Increase the period of  $\tau_s$  to make the taskset schedulable (e.g., by reducing the interference) */
7:     Update  $T_s^l := T_s^c + 1$ 
8:   else
9:     /* Taskset is schedulable with  $T_s^c$  */
10:     $\widehat{T}_s := \widehat{T}_s \cup \{T_s^c\}$  /* Add  $T_s^c$  to the feasible period list */
11:    /* Check schedulability with smaller period for next iteration */
12:    Update  $T_s^r := T_s^c - 1$ 
13:   end if
14: end while
15: Set  $T_s^* := \min(\widehat{T}_s)$  /* Find the minimum feasible period */
16: return  $T_s^*$  /* return the period of  $\tau_s$  */
```

and repeat the search for next security task.

5.5 EVALUATION

I evaluate HYDRA and HYDRA-C on two fronts: (i) a proof-of-concept implementation on an ARM-based rover platform with security applications — to demonstrate the viability of my scheme in a realistic setup (Section 5.5.1); and (ii) with synthetically generated workloads for broader design-space exploration (Section 5.5.2). My implementation is available in a public, open-sourced repository [105].

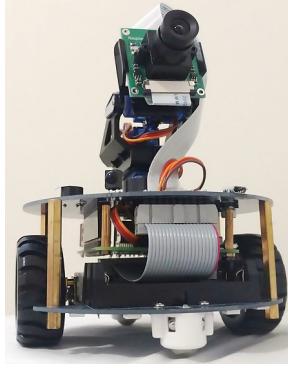


Figure 5.5: Rover used in experiments.

5.5.1 Experiment with an Embedded Platform

I implemented my ideas on a rover platform (Fig. 5.5). The rover peripherals (e.g., wheel, motor, servo, sensor) are controlled by a Raspberry Pi [106] single board computer. I used Linux kernel 4.9 and enabled real-time capabilities by applying the PREEMPT_RT patch [100] (version 4.9.80-rt62-v7+). My experiments were performed on a dual-core setup — this was done by setting the flag `maxcpus=2` in the boot command file `/boot/cmdline.txt`.

In my experiments the rover moved around autonomously and periodically captured and stored images. I assumed implicit deadlines for real-time tasks and considered two tasks: (a) a navigation task — that avoids obstacles using an infrared sensor and navigates (e.g., both driving and path-planning) the rover and (b) a camera task that captures and stores still images. Parameters for the navigation and camera tasks were (C_r, T_r) : (240, 500) ms and (1120, 5000) ms, respectively (i.e., total real-time task utilization was 0.7040).

I introduced two security tasks: (a) an open-source security application, Tripwire [60], that checks intrusions in the image data-store and (b) my custom security task that checks current kernel modules (for detecting rootkits) and compares with an expected profile of modules. I modified Tripwire configurations (`/etc/tripwire/twpol.txt`) and adapted it into periodic execution model. The WCET of the security tasks were 5342 ms and 223 ms, respectively and the maximum periods⁹ of security tasks were assumed to be 10000 ms (e.g., total system utilization is at least $0.7040 + 0.5565 = 1.2605$). The system configurations and tools used in my experiments are summarized in Table 5.1.

Experience and Evaluation

I observed the performance of HYDRA and HYDRA-C by *analyzing how quickly an intrusion can be detected*. I considered the following two realistic attacks¹⁰: (i) an ARM shellcode [97] that

⁹I picked this maximum period value by trial and error so that the taskset became schedulable for demonstration purposes.

¹⁰Note: my focus here is on the integration of any given security mechanisms rather the detection of any particular class of intrusions. Hence I assumed that there were no zero-day attacks and the security tasks were able the detect

Table 5.1: Summary of the Evaluation Platform

Artifact	Configuration/Tools
Platform	1.2 GHz 64-bit Broadcom BCM2837 (Raspberry Pi 3)
CPU	ARM Cortex-A53
Memory	1 Gigabyte
Operating System	Debian Linux (Raspbian Stretch Lite)
Kernel version	Linux Kernel 4.9
Real-time patch	PREEMPT_RT 4.9.80-rt62-v7+
Kernel flags	CONFIG_PREEMPT_RT_FULL enabled
Boot parameters	maxcpus=2, force_turbo=1, arm_freq=700, arm_freq_min=700
WCET measurement	ARM cycle counter registers
Task partition	Linux taskset

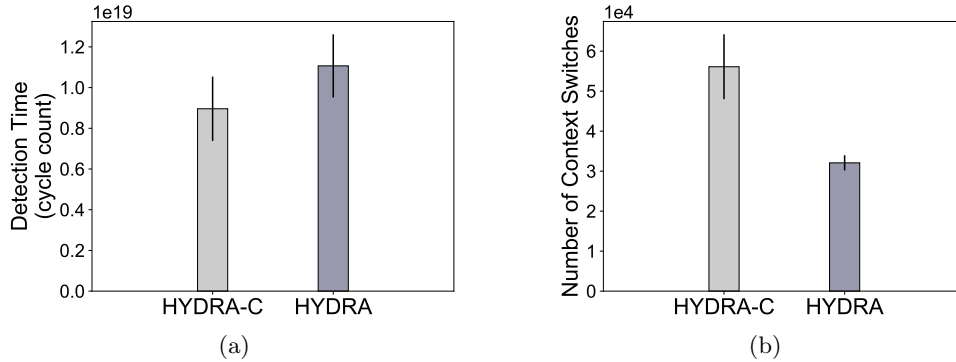


Figure 5.6: Experiments with rover platform: (a) time (cycle counts) to detect intrusions; (b) average number of context switches. On average HYDRA-C can detect the intrusions faster without impacting the performance of real-time tasks.

allows the attacker to modify the contents of the image data-store — this attack can be detected by Tripwire; (ii) a rootkit [107] that intercepts all the `read()` system calls — my custom security task can detect the presence of the malicious kernel module. In Fig. 5.6a I show the average time to detect both the intrusions (in terms of cycle counts, collected from 35 trials) for HYDRA-C and HYDRA schemes. From my experiments I found that, on average, HYDRA-C can detect intrusions 19.05% faster compared to the HYDRA approach (Fig. 5.6a). Since HYDRA-C allows security tasks to migrate across cores, it has shorter periods and that leads to faster detection times.

I next measured the overhead of our security integration approach in terms of number of context switches. For each of the trials I observed the schedule for 45 seconds and counted the number of context switches using the Linux `perf` tool [72]. In Fig. 5.6b I show the number of context switches (y-axis) for HYDRA-C and HYDRA schemes (for 35 trials). As shown in the figure, HYDRA-C increases the number of context switches (since I permit migration across cores). From the corresponding attacks correctly.

my experiments I found that, on average, HYDRA-C increases context switches by 1.75 times. However, this increased context switch overhead *does not impact the deadlines of real-time tasks* (since the security tasks always execute with a priority lower than the real-time tasks) and thus may be acceptable for many real-time applications.

5.5.2 Experiment with Synthetic Tasksets

I also conducted experiments with randomly generated workloads for broader design-space exploration. I considered $M \in \{2, 4\}$ cores and each taskset instance contained $[3 \times M, 10 \times M]$ real-time and $[2 \times M, 5 \times M]$ security tasks. I only considered schedulable real-time tasksets. Each real-time task had periods between $[10, 1000]$ ms and the maximum periods for security tasks were selected from $[1500, 3000]$ ms. I assumed that real-time tasks were partitioned using a best-fit [101] strategy. The utilization of individual tasks were generated using Randfixedsum algorithm [108] and total utilization of the security tasks was at least 30% of the system utilization.

Impact on Schedulability and Security Trade-off

While in this work I consider a legacy system (where real-time tasks are partitioned to cores), for comparison purposes I considered the following two schemes (in addition to HYDRA) that do not consider any period adaptation for security tasks.

- GLOBAL-TMax: In this scheme all the real-time and security tasks are scheduled using a global fixed-priority multicore scheduling scheme¹¹ [23]. Since my focus here is on schedulability I set $T_s = T_s^{max}$, $\forall \tau_s \in \Gamma_S$. This scheme allows me to observe the performance impacts of binding real-time tasks to the cores (due to legacy compatibility).
- HYDRA-TMax: This is similar to the HYDRA approach introduced in Section 5.3 but instead of minimizing periods here we set $T_s = T_s^{max}$, $\forall \tau_s$. This allows me to observe the trade-offs between schedulability and security in a fully-partitioned system.

In Fig. 5.7 I compare the performance of HYDRA-C with the HYDRA, GLOBAL-TMax and HYDRA-TMax strategies in terms of *acceptance ratio* (y-axis) defined as the number of schedulable tasksets over the generated ones. As we can see from the figure, HYDRA-C outperforms HYDRA when the normalized utilization $\frac{\sum C_i/T_i}{M}$ (x-axis) increases. HYDRA-C allows security tasks to execute in parallel across cores and also allocate periods considering the schedulability constraints of all low priority tasks — this results in a smaller response times and can find more tasksets

¹¹I note that there exists recent work [109] that aims to reduce pessimism of multicore schedulability analysis by dividing task WCET into two virtual partitions and then calculating response times by enumerating all possible partitions. Given the workload of real-time and security tasks, my interference calculations (Section 5.4.1) can be adopted to such a two-partitions method. However, from my experiments I found that this extra complexity (i.e., enumerating all WCET partitions) does not improve the schedulability any further.

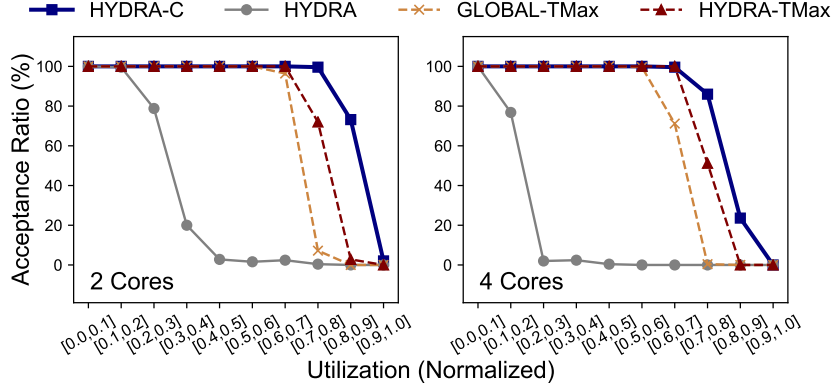


Figure 5.7: Impact on schedulability and security. The acceptance ratio vs taskset utilizations for 2 and 4 core platforms: HYDRA-C outperforms HYDRA and GLOBAL-TMax approaches for higher utilizations.

that satisfy the designer specified bound. In contrast, HYDRA uses a greedy approach that minimizes the periods of higher priority tasks first without considering the global state. Also HYDRA statically binds the security task to the core and hence suffers interference from the higher priority tasks assigned to that core — this leads to lower acceptance ratios. For higher utilizations HYDRA-C can find schedulable tasksets that can not be easily partitioned by using the HYDRA-TMax scheme. The acceptance ratio of HYDRA-C and the HYDRA-TMax scheme is equal when utilization less than 0.7 since some lower priority security tasks experience less interference due to longer periods and specific core assignment. While I bind the real-time tasks to cores (due to legacy compatibility), it does not affect the schedulability since real-time tasks are already schedulable when partitioned and my analysis reduces the interference that real-time tasks have on security ones. I also highlight that while my approach results in better schedulability, HYDRA-C/HYDRA-TMax and GLOBAL-TMax schemes are incomparable in general (i.e., there exists tasksets that may be schedulable by task partitioning but not in global scheme and vice-versa) — I allow security tasks to migrate due to security requirements (e.g., to achieve faster intrusion detection — as I explain in the next experiments, see Fig. 5.8).

In the final set of experiments (Fig. 5.8) I compare the achievable periods (in terms of Euclidean distance) for my approach and the other schemes. The x-axis in the Fig. 5.8 shows the normalized utilizations and the y-axis represents the average difference between the following period vectors $\mathbf{T}^* = [T_s^*]_{\forall \tau_s \in \Gamma_S}$: (a) HYDRA-C and HYDRA (dashed line); (b) HYDRA-C and other strategies (e.g., GLOBAL-TMax and HYDRA-TMax) that do not consider period minimization (dotted marker). Higher distance values imply that the periods calculated by HYDRA-C are smaller (i.e., leads to faster detection time) and HYDRA-C outperforms the other scheme. For low to medium utilizations HYDRA-C performs better when compared to HYDRA. In situations with higher utilizations, the lesser availability of slack time results in HYDRA-C and HYDRA performing in a similar manner. My experiments show that HYDRA-C achieves better continuous monitoring

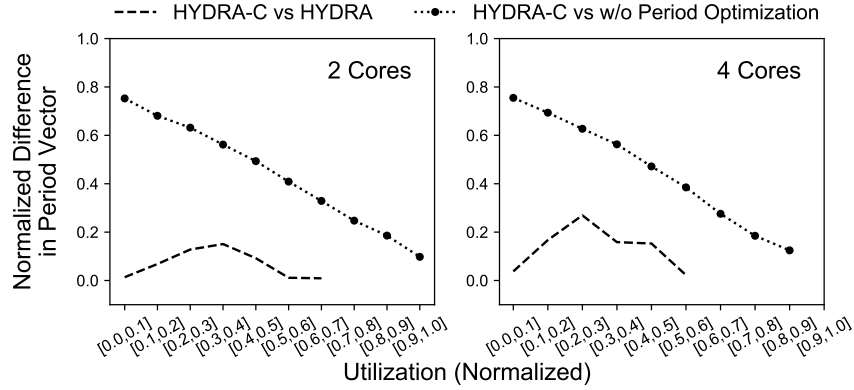


Figure 5.8: Difference in period vectors for HYDRA-C and reference schemes (e.g., HYDRA, GLOBAL-TMax, HYDRA-TMax): the non-negative distance (y-axis in the figure) implies that HYDRA-C finds shorter periods than other schemes.

when compared with both a fully-partitioned approach (HYDRA, HYDRA-TMax) and a global scheduling approach (GLOBAL-TMax) while providing the same or better schedulability.

5.6 CONCLUSION

This chapter presents an evaluation of design choices (HYDRA and HYDRA-C) for integrating security tasks into a multicore RTS. Since I provide comparisons of my solution with two extremes — an static assignment strategy and continuous execution policy that allows runtime migration — this work provides valuable hints to designers on how to build security into such systems. By using my framework, engineers can now evaluate the design choices of such integration techniques to improve the overall security (and hence, safety) of the systems.

In Chapters 4–5 I present pure software-based real-time security integration techniques and assert one part of my dissertation hypothesis (Section 1.1). When both real-time and security tasks are executed on a same platform, an attacker may compromise the underlying host system and prevent the security checks to be successful.¹² In the follow-up work (Chapter 6), I address this issue by leveraging hardware-assisted security features available in modern embedded processors. In particular, I use a trusted (and verified) computing module and execute the security checks inside a secure enclave. This mechanism ensures that even if the (potentially untrusted) real-time tasks are compromised, an adversary can not send false signals. The proposed framework, SCATE (Chapter 6), reason out my dissertation hypothesis since I show the feasibility of integrating security checks by using time-aware hardware/software-based techniques.

¹²I discuss this issue further in Chapter 7.

CHAPTER 6: SELECTIVE CHECKING AND TRUSTED EXECUTION TO PREVENT FALSE ACTUATIONS IN REAL-TIME CYBER-PHYSICAL SYSTEMS

So far I have presented software-based security integration techniques — this asserts one part of my dissertation hypothesis (Section 1.1). I now reason out my hypothesis by presenting a time-aware software/hardware-based real-time security integration framework. In particular, I leverage the capabilities of hardware-assisted security features, viz., the ARM TrustZone [25] and design a targeted security measure (i.e., techniques that prevent falsification of actuation commands). Cyber-physical applications can be vulnerable to attacks that target outgoing actuation commands, i.e., ones that modify the behavior of their physical systems. The limited resources in such systems, coupled with their stringent timing constraints, often prevents the checking of every outgoing command. In this chapter I present a “selective checking” mechanism that uses game-theoretic modeling [110] to identify the right subset of commands to be checked in order to deter an adversary. This mechanism is coupled with a “delay-aware” trusted environment (using TrustZone) to ensure that only verified actuation commands are ever sent to the physical system, thus maintaining their safety and integrity. The *selective checking and trusted execution (SCATE)* framework is implemented on an off-the-shelf ARM platform running standard embedded Linux. I demonstrate the effectiveness of SCATE using four realistic cyber-physical systems (a ground rover, a flight controller, a robotic arm and an automated syringe pump) and study design tradeoffs. Not only does SCATE provide a high level of security that can almost match the performance of a fine-grained mechanism (i.e., one that checks *all* actuation commands), it also suffers significantly lesser overheads (30.48%–47.32% less) in the process. In fact, SCATE can work with more systems than its fine-grained counterpart since the latter causes timing delays that negatively affect the safety of the system. Considering that most CPS do not have any such checking mechanisms, and SCATE is *guaranteed* to meet all the timing requirements (i.e., ensure the safety/integrity of the system), my methods can significantly improve the security (and, hence, safety) of the system.

6.1 INTRODUCTION

Modern cyber-physical applications with real-time requirements are increasingly becoming targets for cyber-attacks. The traditional approaches of air-gapping such systems [13, 18] or using proprietary protocols and hardware [12] have been found wanting in the face of recent high-profile attacks: e.g., Stuxnet [3], attack demonstrations by researchers on medical devices [7] and automobiles [6], denial-of-service (DoS) attacks mounted from IoT devices [111], among others. A common thread among all of the aforementioned attacks, especially ones that threaten the physical safety of the system, is the *falsification of actuation commands* — i.e., commands that control the state of the physical system are either modified or replaced while in transit to the physical component. Note that CPS are comprised of a tight interplay between computation, control

and communication. At its core, a cyber-physical “plant” consists of actuators and sensors that, respectively, monitor and control the physical properties of the system. Due to the tight coupling between actuators and physical plants, such systems are often vulnerable to unexpected situations (e.g., malicious actions) that were not considered during the design/development phases [112]. Hence, sending false/spoofed commands to the actuators can disrupt the normal operation of the physical plant and jeopardize its safety. For instance, an industrial robot on a manufacturing line, must carry out its operation (e.g., placing an object on a conveyor) in 50–100 ms [2]. Failure to do so, could disrupt the entire manufacturing operation and even put the safety of the plant and human operators at risk. If an attacker could modify the command that changes the angle of rotation of the robot arm, the robot may completely miss the conveyor and, potentially, crash the entire system.

In this work I intend to *check actuation commands before they can affect the state of the physical system*. If I find that the command can have negative effects, i.e., it compromises the safety and/or integrity of the CPS, then my proposed technique prevents it from being sent out¹ To prevent tampering, I *implement the checking mechanism in a trusted execution environment (TEE)* that is available on modern commodity processors, viz., the ARM TrustZone [25].²

In an ideal scenario, every outgoing actuation command should be checked. A serious hurdle that prevents such a strategy is that, as mentioned earlier, CPS have stringent timing requirements — very often the actuation command, once sent out, *must be received by the physical system in a short, fixed, amount of time*. This limits the amount of time delays that can be introduced during the checking process. In addition, the control software has its own timing constraints, e.g., it must complete execution before a certain “deadline”; failure to do so can also cause instability in the system. Hence, we *cannot check (and hence, delay) every command* since each check compounds the delays faced by the software. In addition, as seen in Section 6.3.1 the use of the TEE-based checking mechanisms introduces additional delays due to context-switch overheads, further affecting the deadlines. Hence, there is a need to carefully consider *how many* and *which* actuation commands are to be checked — to ensure that (a) the system safety/timing requirements are met and (b) also deter attackers. Picking a fixed subset of actuation commands is not helpful since an adversary can circumvent the checks by targeting the “unchecked” commands. To this end, *I develop a mechanism to validate a random subset of commands, varying at run-time* that significantly increases the difficulty for would-be attackers. I use a *game-theoretic formulation* of a two-player normal-form game [26, 114, 115] for the “selective checking” of the actuation commands (Section 6.4). The combined framework is referred to as the *selective checking and trusted environment (SCATE)* system.

Contributions. In this chapter I make the following contributions:

- I present a framework, SCATE, that protects cyber-physical systems from attacks that falsify

¹Note that the actions taken when we detect a problematic actuation command is orthogonal to the work presented here and depends on the specific CPS. I briefly discuss some of these strategies in Section 7.

²This does not preclude the use of other TEEs, e.g., Intel SGX [113].

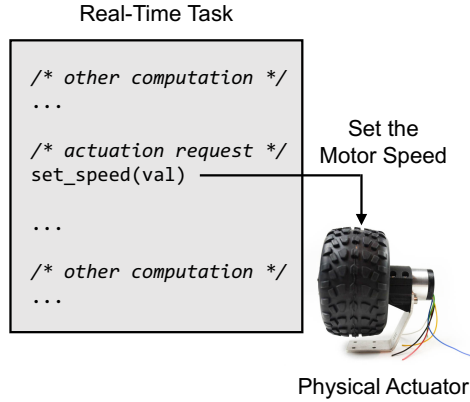


Figure 6.1: Illustration of a vanilla (non-secure) execution scenario that does not check actuation commands.

actuation commands. [Section 6.2.3]

- I use a combination of game-theoretic analysis and a trusted execution environment to deter attackers, significantly reduce checking overheads and still guarantee the safety and integrity of the CPS. [Section 6.4]

I implemented SCATE [Section 6.5] on a commercially available ARM Cortex A53 platform [106] and commodity TEE (TrustZone [25]) running embedded Linux. My system and techniques were evaluated using *four realistic, standardized, cyber-physical systems* [Section 6.5.2]: (a) an autonomous ground rover, (b) a flight controller, (c) a robotic arm (typically found in manufacturing systems) and (d) a syringe pump used in medical devices. I also carried out a broader design space exploration [Section 6.5.3] using simulated workloads and also analyzed the trade-offs between security and timing/safety properties. Not only does SCATE deter attacks, it is able to do so with significantly less overheads and also *guarantee* that it will not compromise the system safety and timing properties. My open-sourced implementation is available in a public repository [116].

6.2 MOTIVATION, OVERVIEW AND BACKGROUND

6.2.1 The Requirement for Checking Actuation Commands

Real-time CPS consists of *cyber* components and *physical* components. The cyber units perform the computing for estimation of the system state (e.g., location of the unmanned vehicle and the direction of its movement) and generation of appropriate control signals for the actuators. The physical components include the entities that are closer to the physical system under observation such as sensors that take measurements and the actuators that mobilize the physical system or its dynamics.

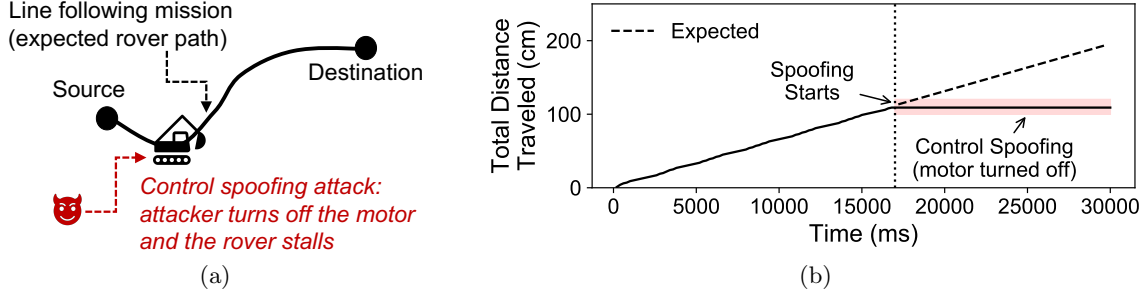


Figure 6.2: Illustration of control spoofing attacks on a rover platform: (a) high-level schematic of my experiment setup where a rover performs a line following mission and an adversary triggers malicious code that turns off one of the rover motors; (b) readings from rover motor encoders under control spoofing attack. Without proper checking, an adversary can inject erroneous signals (shaded region in Fig. 6.2b) and deviate the rover from its expected behavior (dashed line). In this setup the rover stalled in the middle of its mission due to control spoofing attack as shown by the constant readings from wheel motors).

In a non-secure system, when a task generates actuation command, it is being directly issued to the physical actuators. Figure 6.1 illustrates this: when a controller tasks generates an actuation command (to set the speed of the motor) it directly changes the speed without checking whether the speed value is legitimate or not. Without explicit control and verification over actuation process, it is possible to send arbitrary signals to the actuators and an adversary can drive the system in undesirable ways. For instance, consider ground rovers that can be used in multiple cyber-physical applications such as remote surveillance, agriculture and manufacturing [46]. For demonstration purposes, I use a COTS-based ground rover running an embedded variant of Linux on an ARM Cortex-A53 platform (Raspberry Pi [106]). The rover is equipped with two optical encoders that are connected to the motors (i.e., actuator in this setup); it can turn left by switching off the right encoder and vice-versa.

As depicted in Fig. 6.2a, I carried-out a line-following mission where the rover steered from an initial location to a target location by following a line. A controller task runs the standard, pre-packaged, proportional–integral–derivative (PID) closed-loop control [117]. A 5-byte value is sent to the actuator (via memory-mapped registers) to control the wheel motors via the I2C interface [118]. This aids in the navigation and control of the rover. The x-axis of Fig. 6.2 shows the time and the y-axis is the total distance traveled by the rover (i.e., readings from the wheel motors). Since the vendor implementation of the controller does not verify control commands, I was able to inject a logic bomb and send spoofed commands to turn off the motor (marked in the figure). As a result, the rover deviates from its mission. The dashed line (after 17 seconds) shows the expected behavior (viz., without any attack) as a reference (obtained by running a linear regression test from the traces of the uncompromised execution). As the shaded region in the figure shows, the encoder readings (i.e., traversed distance) remained same and the rover was not following the line under the presence of the attack that. I note that designing and scheduling checking techniques for real-time cyber-physical platforms is often more challenging when compared to the general-purpose

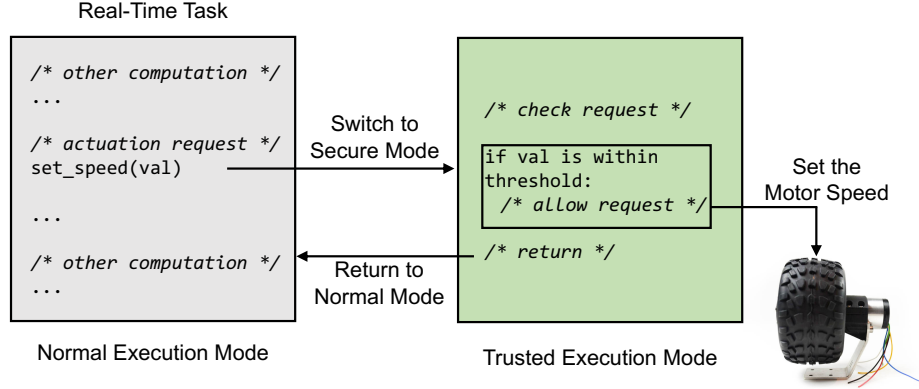


Figure 6.3: Flow of operation in SCATE. When a task generates any actuation command, it will be checked by a trusted entity leveraging TEE technologies (say ARM TrustZone [25]). In this illustration the speed of the actuators (motors attached to the wheels) is only set if the speed value is within a predefined range.

systems due to additional timing/safety constraints (see Table 6.3 for related examples) imposed by such systems. To the best of my knowledge, there is no existing technique that can be directly retrofitted to solve this problem. I believe there is a practical requirement and intellectual merit for designing framework that can protect actuators in real-time CPS and hence is the focus of this research.

6.2.2 System Model

I consider a set of priority-driven, periodic real-time tasks, Γ , running on a multicore CPS platform Π . The set of tasks $\Gamma_p \subset \Gamma$ running on a given core $\pi_p \in \Pi$ is fixed and given by the designers. Each task τ_i issues N_i number of actuation requests. I assume that there is a designer-given quality-of-service (QoS) requirement that $N_i^{min} \leq N_i$ actuation requests (among total N_i number of requests) must be checked (by the trusted entity running inside TEE) for each invocation of a task. I further assume that each actuation command a_i^j is associated a designer-provided weight ω_i^j that represents the importance/preference of checking the corresponding command over other. Higher weight imply that the actuation request is more critical and designers want to examine it more often. I provide a formal representation of my task and real-time model in Appendix C.1.

6.2.3 Problem Overview and Proposed Approach

Actuation commands that are malicious can jeopardize the safety and integrity of cyber-physical applications. In this research I propose techniques to *protect systems from control spoofing attacks by examining actuation commands before* they are being issued to physical peripherals. I name this framework, SCATE where I consider cyber-physical applications consisting of software tasks that can have two different types of execution sections: (a) regular (potentially untrusted) execution

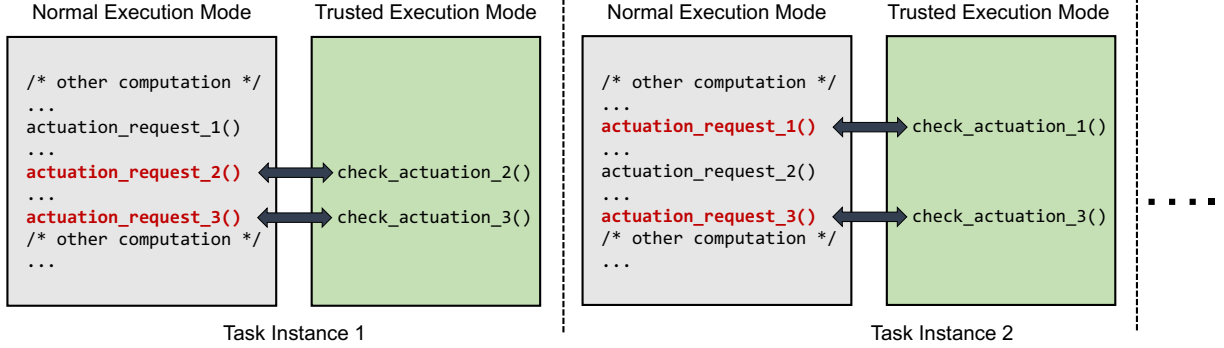


Figure 6.4: Random selection of actuation commands for checking.

section where normal executions are carried-out and (b) trusted sections where critical information (i.e., actuation events) are examined. The high-level schematic of SCATE is depicted in Fig. 6.3. When a task issues actuation command, SCATE transfers the control to secure mode using TrustZone SMC instructions. In the secure mode a designer-given trusted entity checks the actuation commands. In this work I assume that my checking module uses policy rules that defines the mapping of various system states and corresponding legitimate actuation events (refer to Section 6.3 for details). Note that although I use ARM TrustZone as the underlying TEE to demonstrate my ideas, other trusted environments can be used in SCATE without loss of generality.

As we shall see in Section 6.3.1, the context switch overhead for switching between normal and trusted modes is not negligible. For example, consider the rover used in my experiments (Section 6.5). The control loop frequency of the controller task is 5 Hz (200 ms) and it generates four actuation commands (to set the speeds and direction of attached motors). The controller task must complete execution before its sampling interval (200 ms). If we check the speed and direction values of each of the four commands using TrustZone, the controller task fails to comply with its timing requirements (since it requires 261 ms to finish). For such situations (i.e., when not all the commands can be verified while respecting timing guarantees for all the tasks), SCATE *selectively checks a subset of actuation events*. For this, I leverage the tools from *game theory* [26, 114] and randomly select a subset of commands (for checking) in each job of a task that provides us a trade-off between security and timing guarantees (see Section 6.4 for details). Figure 6.3 shows an illustrative case where a tasks generates three actuation requests and we can check at most two request to comply with timing/safety requirements. In this case SCATE randomly checks two commands in different task instances (e.g., command (2,3) in first instance and command (1,3) in second instance). From earlier rover example, by reducing the number of checks to half³ (e.g., randomly checks two commands in each task instances instead of all four), SCATE manages to finish the controller task before 200 ms without significantly degrading security (e.g., on average, SCATE requires one additional task instances to detect an attack when compared to the scheme that checks all four commands).

³Section 6.5 presents details of this experimental setup and additional results.

6.2.4 Background

I now provide background on the TEE technology (ARM TrustZone [25]) and the game-theoretic modeling tool [26]) used in this work.

Trusted Environments and ARM TrustZone

Trusted environments are set of hardware and software-based security extensions where the processors maintain a separated subsystem in addition to the traditional OS (also called rich OS) components. TEE technology has been implemented on commercial hardware such as ARM TrustZone [25] and Intel SGX [113]. In this work I consider TrustZone as the building block of our model due to wide usage of ARM processors in CPS. I note that although I use the TrustZone functionality for demonstration purposes, my ideas are rather general and can be adapted to other TEE technology without loss of generality.

TrustZone contains two different privilege blocks: *(i)* regular (non-secure) execution environment, called “Normal World” (NW) and *(ii)* a trusted environment, referred to as “Secure World” (SW). The NW is the untrusted environment that runs a commodity untrusted OS (called rich OS) whereas SW is a protected computing block that only runs privileged instructions. TrustZone hardware ensures that the resources in the SW can not be accessed from the NW. These two worlds are bridged via a software module, the *secure monitor*. The context switch between the NW and SW is performed via a *secure monitor call* (SMC).

Normal-Form Games

The overheads for TEE context switch is costly (Section 6.3.1). *If a task cannot verify all the actuation commands, I propose to select only a subset of commands in each job for checking.* For this, I leverage the tools from game theory [110] to ensure that the chosen subsets are non-deterministic, at least from the adversary’s point of view (see Section 6.4 for details). In multi-agent systems, if the optimal action for one agent to take depends on the actions that the other agents take, game theory is used to analyze how an agent should behave in such settings. In a *normal-form game* [26], every player $j \in \{1, 2, \dots, J\}$ has a set of strategies (or actions) σ_j and a utility function $u_j : \sigma_1 \times \sigma_2 \times \dots \times \sigma_J \rightarrow \mathbb{R}$ that maps every outcome (a vector consisting of a strategy for every player) to a real number. As we shall see in Section 6.4.1, I formulate this problem as a *two-player game* (e.g., system designer and adversary). The output of the game finds the probability distribution over the player’s strategies (i.e., fraction of time a given strategy is selected in the game) that leads to optimal outcome. While game-theoretic analysis has been used in other modeling problems (e.g., patrolling [115], network routing [119], transportation systems [120], tracking information flows [48], decision making [50]), to the best of my knowledge this is the first work that uses normal-form games in the real-time security context.

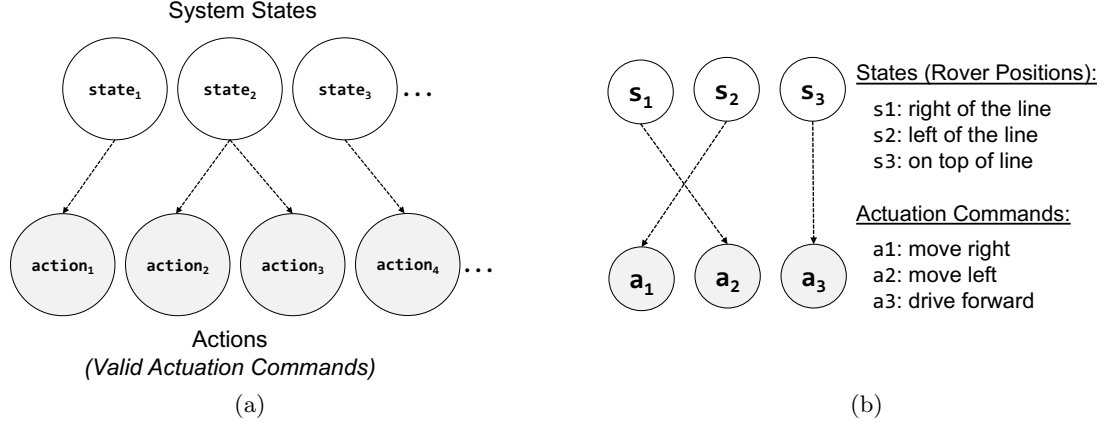


Figure 6.5: (a) $State \rightarrow Action$ mapping used in SCATE for checking actuation commands. (b) Example states and corresponding valid actuation commands for a line-following rover.

6.3 CHECKING ACTUATION COMMANDS

In SCATE a “checking module” executes inside the trusted environment. The checking module observes system states and decides whether a given actuation command is legitimate or malicious. Recall that when a task issues actuation requests, SCATE transfers control to the secure execution mode using TrustZone SMC instructions. In particular, for a given real-time platform I assume that there exists a $\text{CheckAct}(\tau_i, a_i^j, t)$ function⁴ that examines a given actuation command a_i^j (where $1 \leq j \leq N_i$, N_i is total number of commands the task issues) generated by a task τ_i at a given time t . As shown in Fig. 6.5a, the checking module uses policy abstraction rules [121], viz., $State \rightarrow Action$ pairs where the $State$ predicate represents a given system state and $Action$ denotes corresponding valid actuation command(s). In particular, I assume that when τ_i executes an actuation command, function $\text{CheckAct}(\tau_i, a_i^j, t)$ first observes system state $\mathcal{S}(t)$ and then decides whether the actuation command a_i^j is valid for the current state $\mathcal{S}(t)$. For instance, consider the line-following rover presented in Section 6.2.1. The directions for the wheels of the rover (i.e., forward, left and right; controlled by the attached motors) are the actuation commands. At any given point of time, the rover can be in one of three states: $\mathcal{S} = \{\text{ON_LINE}, \text{LEFT_OF_LINE}, \text{RIGHT_OF_LINE}\}$ that denotes whether the rover is on top of the line or shifted left/right of the line, respectively. The rover controller task performs the following actuation operations (i.e., actions): `move_left()/move_right()` (move the rover left/right, respectively) and `move_forward()` (drive the rover forward). The corresponding $State \rightarrow Action$ mapping for this rover is illustrated in Fig. 6.5b. If the rover is on left side of the line (i.e., $State = \text{LEFT_OF_LINE}$), the valid command should be `move_right()` (i.e., shift the rover back to the line so that it stays on the line) and if the rover is on top the line (i.e., $State = \text{ON_LINE}$), the controller task should drive it forward (i.e., issue `move_forward()` command).

Table 6.1 summarizes the possible checks for various real-time platforms — however, this is by

⁴The exact function depends on the specific CPS and application requirements.

Table 6.1: Actuation Command Checking for Various Cyber-Physical Platforms*

Platform	Application	Actuators	Possible Checking Conditions
Robotic vehicle (ground, aerial)	Surveillance, agriculture, manufacturing	Servo, motor	(a) Check if the robot is following the mission; (b) allow only predefined number of actuation commands per period
Robotic arm	Manufacturing	Servo, buzzer	(a) Check the servo pulse sequences matches with the desired (design-time) sequence; (b) do not raise alarm if the pulse sequence is normal
Infusion/Syringe pump	Health-care	Motor, display	(a) Drive the motor only to allowable positions/rates (b) display only the amount of fluid infused (e.g., obtained from motor encoders)
Water/air monitoring system	Home/industrial automation	Buzzer, display	(a) Send high pulse to buzzer only if water-level is high/air quality abnormal/detect smoke; (b) do not display alert if the system state is normal
Surveillance system	Home/industrial automation	Servo, buzzer	(a) Trigger alarm only if there is an impact/object detected in camera; (b) rotate camera (using servos) only within allowable pan/tilt angle

*Platforms listed in shaded rows are implemented and evaluated in this work. Other examples are presented here to illustrate applicability of my ideas for multiple use-cases.

no stretch meant to an exhaustive list. I assume that $State \rightarrow Action$ rules are given by the designers based on system requirements. I note that the ideas presented here are agnostic to the specific checking method and SCATE is compatible with existing techniques (e.g., defining rules at design times [62, 122], deriving from specifications [123] and based on statistical analysis [74].) In this work I focus on how to selectively examine random subsets of actuation commands by using designer-provided checking rules. In Section 6.5.2 I describe implementation of `CheckAct()` functions for four realistic platforms used in my evaluation.

6.3.1 The Requirement for Coarse-Grain Checking

In order to check actuation commands we must ensure that SCATE should not cause inordinate delays and the timing requirements of real-time tasks are satisfied (i.e., they complete execution before deadlines). I therefore develop design-time tests (see Appendix C.2) that ensure tasks meet their timing requirements (deadlines). My analysis in Appendix C.2 shows that there is an overhead for inspecting the actuation commands using TEEs and a task may miss its timing/safety requirements. As I mentioned earlier, any failure to meet timing requirements disrupts the stability of the system and can be catastrophic. For instance, consider the rover example from Section 6.2.3. If the controller task fails to issue navigation commands before the time limit, the rover may stall, or worse, may not be able to steer properly and even crash. Hence, delays caused by the checking mechanisms can also result in such problems. Table 6.3 presents additional examples for the possible consequences when tasks are unable to meet their timing requirements.

Existing work [124–126] show that although TEEs are implemented on hardware, they can still cause significant overheads — this is particularly acute in real-time applications. For instance, consider the Linux-based TrustZone port, OP-TEE [127] supported on many embedded platforms.

My experiments show that the overhead of switching between normal to trusted mode is around 66 ms for Raspberry Pi platform. For completeness, I also performed experiments on an ARMv8-M Cortex-M33 architecture using ARM FVP libraries [128] where the regular applications were running on FreeRTOS [94] and trusted mode codes were executed on bare-metal. I find that the mode switching delays in this setup are 2 ms. I note that the delays are higher in Linux environment due to extra overheads (i.e., execution of sequence of API calls [124]) imposed by Linux kernel and OP-TEE secure OS. Although the overheads of secure calls (SMC) for switching between regular and trusted modes are platform-specific, it may not be feasible to check multiple actuation commands while retaining real-time guarantees. For example, if a task operates at 50 Hz (i.e., required to finish before 20 ms) [63] and regular computation takes 10 ms, the FreeRTOS-based setup allows at most 5 checks to comply with timing requirements. Likewise, for applications running on a Linux and OP-TEE-based Raspberry Pi platform (Section 6.5.2), we can check at most 3 commands per instance if the controller task operates at 5 Hz. We therefore need smart techniques, say where *only a subset of commands are vetted for each instances while maintaining security guarantees*, to support TEE-based checking for real-time applications. I now present my methods to achieve this (based on game-theoretic analysis) in the following section.

6.4 GAME-THEORETIC ANALYSIS FOR RANDOM CHECKING

As checking all (or most of the) actuation commands can jeopardize the safety of the real-time tasks, I now propose a mechanism to deal with this issue of monitoring overheads. I consider the case when there exists a task τ_i such that τ_i cannot perform all the N_i checking before its deadline (denoted by D_i). One option to reduce the number of checks is to verify only a subset of commands so that the task can finish before its deadline. That is, check a subset of commands, K_i , ($N_i^{min} \leq K_i < N_i$) such that $R_i^{TEE} \leq D_i, \forall \tau_i \in \Gamma$ where R_i^{TEE} is the response time (i.e., time between task arrival to completion). The challenge is then to decide *which* subset of K_i (among N_i) actuation requests should be selected for checking in each task τ_i . In addition, if we check only fixed K_i commands and an adversary jeopardizes some or all of the remaining $N_i - K_i$ requests, then the attack will succeed and remain undetected. To balance the security and real-time requirements, SCATE *randomly selects different subsets of requests for checking*. In particular, at each task instances SCATE randomly picks a set of K_i commands with pre-computed probability distributions. While I pick a subset of commands, it should look like (to adversary) that SCATE is checking it all. As a result, it will be difficult for an attacker to identify which subset of requests are selected for checking in evading detection and thus less chances of success — since each instance of a task will select a different subset of requests for vetting. As we shall see in Section 6.4.1 I formulate this problem as a two-player game [26] and develop Algorithm 6.1 to determine the feasible number of K_i inspection points that provide similar level of security. I now illustrate my ideas with a simple example.

Intuition and Example. Let us consider a ground rover performing a line-following mission. The rover controller task (τ_c) generates the following actuation requests ($N_c = 3$): (a) $\text{setEnc}_L(val)$ and $\text{setEnc}_R(val)$ that set the speed of left and right motor encoders, respectively (denoted by a_c^1 and a_c^2); (b) $\text{setNav}(cmd)$ that issues a navigation command where each cmd specifies values to the peripheral registers for navigating the rover forward, backward, left and right direction (denoted by a_c^3). Recall from the description of the system model that there exists designer-provided weights ω_i^j for checking each of the commands a_i^j (see Appendix C.1). Let the weights are given by: $\Omega_c = \{\omega_c^1, \omega_c^2, \omega_c^3\}$. As we shall see in Section 6.4.1, the weights are used to determine which commands will be checked more often. For example, if $\omega_c^3 = 2$ and $\omega_c^1 = \omega_c^2 = 1$, then SCATE tends to check $\text{setEnc}_R(val)$ twice more times that the other two commands. The checking for a_c^1 and a_c^2 involves to check whether the speed value is within a given bound (e.g., $val \in [v^-, v^+]$) and for a_c^3 the checking module verifies if the cmd value is consistent so that the rover is within the line and correctly follows the mission. I now consider the case when checking all three requests does not comply with the timing requirement of τ_c and we can only verify at most $K_i = 2$ requests (I describe how to calculate the value of K_i for each task τ_i in Section 6.4.2). Therefore, the possible combinations for perform checking are as follows: $X_c = \{(a_c^1, a_c^2), (a_c^2, a_c^3), (a_c^1, a_c^3)\}$. For each instance of the task τ_c , SCATE randomly selects any j -th element from the set X_c with probability x_c^j that provides better “monitoring coverage”. For example, let $x_c^1 = x_c^2 = 0.25$ and $x_c^3 = 0.5$. Then, for any given instance of τ_c the possibility of verifying both a_c^1 and a_c^2 is 25%, verifying both a_c^2 and a_c^3 is 25%, where the possibility of verifying a_c^1 and a_c^3 is 50% (recall that by assumption we can check only 2 commands per job). In the following section I present my ideas to compute these probabilities using game-theoretical analysis.

6.4.1 Generating Randomized Schedules

I now present our techniques to derive the probabilities for selecting random subsets of commands to be checked. The formulations this section assumes that the size of feasible subset of commands K_i — that ensures all timing requirements are met — is known for each task. In Section 6.4.2 I present algorithms to derive K_i . I model the selection of a subset of commands (for checking) as *two-player normal-form game* (also called leader-follower game) [26, 114, 115]. The game formulations allow the designers to model the fact that an attacker acts with knowledge of defender’s actions and thus reacting accordingly. Since normal-form games address the challenges posed by my context (i.e., selecting optimal actions for decision-making agents), I use this game model in SCATE for generating randomized schedules.

Game Setup

In my model I consider two players: the system designer and the adversary. Let X_i denote the set of all combinations of choosing K_i subset of commands from total N_i number of possibilities,

i.e., the size of set $|X_i| = \binom{N_i}{K_i} = \frac{N_i!}{K_i!(N_i-K_i)!}$. In game-theoretic terminology X_i represents the set of designer’s “strategies”. As we discuss in the previous section, the j -th element of X_i is a vector of size K_i that represents which subset of commands will be picked for inspection. Let us now introduce the variable Q_i that represents the attacker’s set of actions. The set Q_i represents the possible combinations of actuation requests invoked by τ_i that an adversary can compromise. Recall from the rover example where the controller task τ_c invokes $N_c = 3$ actuation requests, the adversary can pick one of the following eight combinations: $Q_c = \{(a_c^1), (a_c^2), (a_c^3), (a_c^1, a_c^2), (a_c^2, a_c^3), (a_c^1, a_c^3), (a_c^1, a_c^2, a_c^3), (\emptyset)\}$. For example, the first element in the set denotes the adversary chooses to compromise only invocation a_c^1 , the fifth element implies both a_c^2 and a_c^3 are compromised while the last element implies there is no attack in the job during this instance of the task. Note that the size of the attacker’s strategy set Q_i is 2^{N_i} .

Recall that each actuation command a_c^j is associated with a designer-given weight ω_c^j (see Section 6.2.2 and Appendix C.1). The higher weight for a given command implies that designers want to check the corresponding command often. For instance, from the rover example designers may want to check navigation commands (a_c^3) more frequently than the ones that set the wheel speeds (a_c^1, a_c^2) and may set higher weight for ω_c^3 . Let $\Lambda(X_i^j)$ denote the set of commands used for vetting and $\Psi(X_i^j)$ is the set of corresponding weights in the j -th element of the strategy set X_i . Likewise $\Lambda(Q_i^l)$ denotes the set of commands and $\Psi(Q_i^l)$ is the corresponding set of weights compromised by the attacker in its l -th strategy. In the rover example, if we select $j = 2$ and $l = 4$ (i.e., second and fourth elements of the designers and adversary’s strategy set) then $\Lambda(X_c^2) = \{a_c^2, a_c^3\}$, $\Psi(X_c^2) = \{\omega_c^2, \omega_c^3\}$ and $\Lambda(Q_c^4) = \{a_c^1, a_c^2\}$, $\Psi(Q_c^4) = \{\omega_c^1, \omega_c^2\}$. I now introduce two variables, viz., *system reward* (λ) and *system cost* (ζ). *Higher system reward and lower cost is good for the designers and bad for the attackers*. Likewise, higher system cost and lower reward is favorable for the adversary’s point of view (and bad for the designers).

If a task τ_i selects the j -th element from set of strategies X_i and the attacker selects the l -th strategy from Q_i for attack then the system reward is $\lambda_i^{j,l}$ and cost is $\zeta_i^{j,l}$. If the task selects a subset of commands for vetting in its j -th strategy and the adversary also attacks those invocations in its l -th strategy, i.e., $\Lambda(X_i^j) = \Lambda(Q_i^l)$, it implies that the attack is detected. Hence, I set $\lambda_i^{j,l}$ a large positive value (i.e., high system reward, since the attack is detected) and $\zeta_i^{j,l}$ a large negative value (i.e., no system cost). In contrast, if $\Lambda(X_i^j) \cap \Lambda(Q_i^l) = \emptyset$ for any pair (j, l) , i.e., $\Lambda(X_i^j)$ does not contain any commands in attackers l -th strategy $\Lambda(Q_i^l)$, that implies the compromised commands are not vetted (i.e., the intrusion is not checked). In this case I set $\lambda_i^{j,l}$ a large negative value (i.e., no reward) and $\zeta_i^{j,l}$ a large positive value (i.e., high system cost).

When the above two conditions do not hold (i.e., only a subset of the compromised commands are checked) and therefore $\exists(j, l)$ such that $\Lambda(X_i^j) \cap \Lambda(Q_i^l) \neq \emptyset$, I then obtain the system reward/cost by *normalizing the weights* of both adversary and designer’s strategies. For this, I define the reward and cost functions as follows.

System Reward:

$$\lambda_i^{j,l} = \frac{\sum_{w \in \Psi(X_i^j)} w}{\sum_{w \in \Psi(X_i^j) \cup \Psi(Q_i^l)} w} \quad (6.1)$$

System Cost:

$$\zeta_i^{j,l} = \frac{\sum_{w \in \Psi(Q_i^l)} w}{\sum_{w \in \Psi(X_i^j) \cup \Psi(Q_i^l)} w}. \quad (6.2)$$

Let me revisit the rover example with $j = 2$ and $l = 4$. In this case $\lambda_c^{2,4} = \frac{\omega_c^2 + \omega_c^3}{\omega_c^1 + \omega_c^2 + \omega_c^3}$ and $\zeta_c^{2,4} = \frac{\omega_c^1 + \omega_c^2}{\omega_c^1 + \omega_c^2 + \omega_c^3}$. This reward and cost functions give me one way to measure the security of the system in terms of how many significant invocations SCATE can monitor given an attacker's strategy. Higher system reward (and lower cost) implies that SCATE performs more checking with respect to a given adversarial action.

Formulation as an Optimization Problem

I now develop models to determine the optimal strategy for each of the tasks. Let me now denote x_i^j is the probability of selecting j -th element from X_i (represents the proportion of times in which a strategy j is used by the task τ_i in the game). The output of the game will provide us the probability distribution of (randomly) selecting subset of K_i commands from the set possible choices (i.e., X_i) for the different instances of a given task τ_i . For a given adversarial strategy l , summing over all the strategy sets X_i (i.e., $\sum_{j=1}^{|X_i|} x_i^j \lambda_i^{j,l}$ and $\sum_{j=1}^{|X_i|} x_i^j \zeta_i^{j,l}$) gives us the total system reward and cost, respectively.

We can obtain probability distributions of selecting elements from X_i for a given attacker strategy l (that maximizes the system reward) by forming a linear optimization program. In particular, for each of the attacker's l -th strategy (where $1 \leq l \leq |Q_i|$), I compute a strategy for the τ_i such that (i) playing l -th strategy is a best response from the adversary's point of view (i.e., more system cost) and (ii) under this constraint, the strategy maximizes the reward for τ_i (i.e., checks critical commands more often). The linear optimization program is given as follows.

$$\max_{x_i^j} \sum_{j=1}^{|X_i|} x_i^j \lambda_i^{j,l} \quad (6.3)$$

$$\text{Subject to: } \forall l' \in [1, |Q_i|], \sum_{j=1}^{|X_i|} x_i^j \zeta_i^{j,l} \geq \sum_{j=1}^{|X_i|} x_i^j \zeta_i^{j,l'} \quad (6.4)$$

$$\sum_{j=1}^{|X_i|} x_i^j = 1 \quad (6.5)$$

$$x_i^j > 0, \forall j \in [1, |X_i|]. \quad (6.6)$$

The objective function in Eq. (6.3) maximizes the total system reward. The constraint in Eq. (6.4) ensures that the current (e.g., l -th) strategy results in higher cost for the attacker when compared to other adversarial strategies. The constraint in Eq. (6.5) ensures the sum of probability distributions equal to unity and the last constraint in Eq. (6.6) ensures non-zero probabilities so that all combinations of the actuation commands from X_i can be selected.

Let $[x_i^j]_{j=1:|X_i|}(l)$ denote the solution obtained from the linear programming formulation for the l -th adversarial strategy. Then, from all feasible strategies l (where $1 \leq l \leq |Q_i|$) I choose the one (say l^*) that maximizes the objective value in Eq. (6.3), i.e., $l^* = \operatorname{argmax}_{1 \leq l \leq |Q_i|} \sum_{j=1}^{|X_i|} x_i^j \lambda_i^{j,l}$.

The variables $[x_i^j]_{j=1:|X_i|}(l^*)$ obtained by solving the corresponding linear program gives us the probability distributions of selecting K_i subset of commands from total N_i commands. The game-theoretical analysis allows me to show that the probability distributions obtained by solving the l^* -th linear program will be optimal for the task τ_i (i.e., maximizes system reward) [26, 115].

For a given strategy l , the above linear programming formulation can be solved using standard off-the-shelf optimization solvers [129, 130] in polynomial time. Since the strategy set Q_i is finite by definition, we can calculate the optimal probability distributions (i.e., $[x_i^j]_{j=1:|X_i|}(l^*)$) in finite amount of time since it is polynomial in the total number of adversarial strategies.

6.4.2 Calculating the Size of the Feasible Command Set

My focus here is to *examine as many actuation commands as possible* while meeting real-time guarantees. The game formulation from the previous sections assumes that I know the size of the set K_i and calculate the probabilities accordingly. However, in a system with multiple real-time tasks, finding the size of feasible set K_i for each task $\tau_i \in \Gamma$ while also meeting the real-time requirements (deadlines) is a non-trivial problem. I therefore develop an iterative solution for finding the size of this set.

My proposed solution works as follows (refer to Algorithm 6.1 for a formal description). In Lines 1–4, I first fix $K_i = N_i^{\min}, \forall \tau_i$ and check the whether all tasks meet their timing requirements (i.e., finish before their deadlines). If there exists a task such that it fails to meet timing requirements, I report that it is “*infeasible*” to integrate SCATE in the target system while satisfying designer specified QoS requirements (Line 7). This infeasibility result provides hints to the designers to update or modify system parameters (e.g., number of commands, QoS requirements) to enable the ability to check actuation commands in the system. Otherwise (i.e., $R_i^{TEE} \leq D_i$), I optimize the number of commands a task can verify in an iterative manner (Lines 9–18). To be specific, for a given task τ_s I perform a logarithmic search (see Algorithm 6.2 for the pseudo-code) and find the maximum number of commands K_i^* that can be verified within the range $[N_i^{\min}, N_i]$ such that all low-priority tasks τ_l meets their timing requirements (Line 11). If the selected parameter K_i^* is less than the total commands N_i , I then use game theoretical-analysis from Section 6.4.1 and obtain probabilities of randomly selecting K_i^* commands (in each task instance) from a total of

Algorithm 6.1: SCATE: Parameter Selection

Input: Input taskset parameters Γ

Output: For each task τ_i , the size of the feasible set $K_i^* \geq N_i^{min}$ and selection probability $x_i^j, j = 1, \dots, |X_i^*|$ for each of the combinations in the strategy set X_i^* ; Infeasible otherwise.

```
1: /* Check minimum feasibility requirements */
2: for each  $\tau_i \in \Gamma$  do
3:   Set  $K_i = N_i^{min}$  and calculate response time  $R_i^{TEE}$  using Eq. (C.3)
4: end for
5: /* Unable to integrate SCATE with minimum QoS requirements */
6: if  $\exists \tau_i$  such that  $R_i^{TEE} > D_i$  then
7:   return Infeasible
8: end if
9: for each task  $\tau_i$  (from higher to lower priority order) do
10:  Find maximum  $K_i^* \in [N_i^{min}, N_i]$  such that all low-priority tasks  $\tau_l$  meet their timing requirements (i.e.,  $R_l^{TEE} \leq D_l$ )
11:  /* not all the commands can be examined — obtain parameters for non-deterministic checking */
12:  if  $K_i^* < N_i$  then
13:    Determine the strategy set  $X_i^*$  for  $K_i^*$  where  $|X_i^*| = \binom{N_i}{K_i^*}$  and
    obtain probabilities  $x_i^j$  by solving the game formulation
14:  end if
15:  Update response time  $R_l^{TEE}$  for each  $\tau_l$  that executes with a priority lower than  $\tau_i$ 
    with the updated size  $K_i^*$ 
16: end for
17: /* return the solution */
18: return the size of the feasible set  $K_i^*$  and
    probability  $x_i^j$  of selecting  $j$ -th strategy ( $j = 1, 2, \dots, |X_i^*|$ ) from  $X_i^*$ 
    for each task  $\tau_i \in \Gamma$ 
```

Algorithm 6.2: Calculation of Maximum Feasible Actuation Requests for a Given Task

τ_i

```
1: Define  $K_i^l := N_i^{min}$ ,  $K_i^r := N_i$ ,  $K_i^c := 0$ 
2: Set  $\hat{\mathcal{K}}_i := \{N_i^{min}\}$  /* Initialize a variable to store feasible values */
3: while  $K_i^l \leq K_i^r$  do
4:   Update  $K_i^c := \lfloor \frac{K_i^l + K_i^r}{2} \rfloor$ 
5:   if  $\exists \tau_l \in lp(\tau_i, \pi_p)$  such that  $\tau_l$  is not schedulable with  $K_i = K_i^c$  then
6:     /* Decrease verification load to make the taskset schedulable */
7:     Update  $K_i^r := K_i^c - 1$ 
8:   else
9:     /* Taskset is schedulable with  $K_i^c$  */
10:     $\hat{\mathcal{K}}_i := \hat{\mathcal{K}}_i \cup \{K_i^c\}$  /* Add  $K_i^c$  to the feasible list */
11:    /* Check schedulability with larger  $K_i$  for next iteration */
12:    Update  $K_i^l := K_i^c + 1$ 
13:  end if
14: end while
15: /* return the maximum from the set of feasible values */
16: return  $\max(\hat{\mathcal{K}}_i)$ 
```

N_i commands (Line 14). The above process is repeated for all the tasks and the algorithm finally returns the corresponding selection probabilities (Line 20).

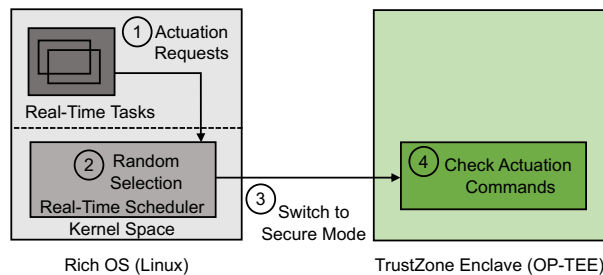


Figure 6.6: Sequence of steps involved in the SCATE operation: when the tasks generate actuation requests (e.g., `ioctl()` calls), the commands are first received by the scheduler and then (randomly) selected requests are transferred to the secure enclave for checking.

6.5 EVALUATION

In this section I first present my implementation details (Section 6.5.1). I then show the viability of SCATE (*i*) using four realistic cyber-physical case-studies (Section 6.5.2) and (*ii*) generated workloads for a broader design-space exploration (Section 6.5.3).

6.5.1 Implementation

I implemented SCATE on Raspberry Pi 3 Model B [106] (equipped with 1.2 GHz 64-bit ARMv8 CPU and 1 GB RAM). I selected Raspberry Pi as my implementation platform since (*a*) it supports a commodity TEE (ARM TrustZone), (*b*) existing literature [15, 45, 62, 131–133] has shown the feasibility of deploying cyber-physical applications on Raspberry Pi and (*c*) it provides a robust development environment that allows me to analyze the viability of my approach on multiple realistic off-the-shelf cyber-physical systems under a common platform. In my experiments I considered both motors (DC as well as stepper) and servos as actuators. I used the Adafruit motor shield [134] (an I/O extension daughter-board for Raspberry Pi) that allowed us to control multiple actuators using the I2C interface. For controlling motors and servos I used an open-source motor driver [135] and servo controllers [136]. I implemented trusted execution modes using the OP-TEE [127] software stack that uses GlobalPlatform TEE APIs [137]. OP-TEE provides a minimal secure kernel (called OP-TEE core) that can be run in parallel with a rich OS (e.g., Linux). I used Ubuntu 18.04 filesystem with a 64-bit Linux kernel (version 4.16.56) as the rich OS and executed `CheckAct()` functions in the OP-TEE secure kernel (version 3.4). The controller and checker codes are written in C for compatibility with the OP-TEE APIs. For accuracy of my measurements I disabled all the frequency scaling features in the kernel and executed RP3 at a constant frequency (i.e., 1.2 GHz, the maximum supported clock speed). This was to ensure that values observed in different trials were consistent.

The linear programs were solved using the Python-MIP library [130] with CBC solver [138]. SCATE operates at the scheduler-level (see Fig. 6.6). When a task generates actuation requests (i.e., `ioctl()` calls, see block ① in Fig. 6.6), the real-time scheduler randomly selects a subset

Table 6.2: Summary of My Implementation Platform

Artifact	Configuration
Platform	Broadcom BCM2837 (Raspberry Pi 3)
CPU	1.2 GHz 64-bit ARM Cortex-A53
Memory	1 Gigabyte
Operating System	Linux (NW), OP-TEE (SW)
Kernel version	Linux kernel 4.16.56, OP-TEE core 3.4
Interface	I2C
Boot parameters	dtparam=i2c_arm=on, dtparam=spi=on, force_turbo=1, arm_freq=1200, arm_freq_min=1200, arm_freq_max=1200





of commands (block ②). In particular, from the probabilities obtained by the game model (Section 6.4.1), SCATE uses the roulette-wheel selection technique [139] for selecting a random subset of commands at runtime (i.e., for each instance of a task). My implementation uses a standard C random number generator. However, this does not preclude the use of other hardware-supported generators such as Z1FFER [140] and OneRNG [141] to ensure tamper-proof true random number generation and further improve the security of SCATE. For each of the selected commands, the scheduler then transfers the control to the secure enclave (i.e., OP-TEE) for checking (blocks ③ and ④ in Fig. 6.6). For each of my case-studies, I implemented the `CheckAct()` as an OP-TEE trusted application. The implementation and details of `CheckAct()` for the each of platforms is presented in Section 6.5.2. I note that my implementation using Raspberry Pi, Linux and OP-TEE serves as a good proof-of-concept and can be extended with other OS, hardware platforms and TEE architecture without loss of generality. My implementation code is available in a public repository [116]. Table 6.2 summarizes system configurations and implementation details.

6.5.2 Experiments with Realistic Cyber-Physical Platforms

I chose four realistic real-time cyber-physical platforms as case-studies to evaluate the efficacy of SCATE: (a) ground rover, (b) UAV flight controller, (c) robotic arm and (d) syringe/infusion pump that are used in many cyber-physical applications. These are off-the-shelf platforms and I did not modify them. I note that unlike generic applications, there are few publicly available open-source real-time platforms due to their proprietary nature (see more in Chapter 7). In addition, there is a significant amount of effort involved in setting up a TEE-supported real-time cyber-physical platform and generating evaluation traces from it. I therefore limit ourselves to four real-time platforms in this paper — albeit they cover a wide range of application domains (see Table 6.1) and should suffice to demonstrate the feasibility of my approach.

Since I focus on scheduling independent actuation checking events, the type of attacks and checking techniques are orthogonal to my model. For demonstration purposes, I use fault injection [142, 143] to mimic malicious behavior and trigger attacks that are known to the checking module (i.e., `CheckAct()` function). Note that this is a standard technique used by the researchers

Table 6.3: Real-Time Cyber-Physical Platforms used in My Experiments

Platform	Application	Real-Time/Safety Requirements	Actuation Commands	Attack Demonstration	Checks inside Enclave
	The rover performs a line following mission. The controller task sets the speed of the rover and steers the wheels (based on its position on the line) by executing a PID control loop	Set the speed and direction of the motors for the wheel movements within sampling interval (i.e., control loop frequency, set at 5 Hz)	<ul style="list-style-type: none"> Set the speed of the wheels Set wheel directions (left, right, forward and backward) 	DoS attack [62]: arbitrarily sets high speed for one of the wheel motors	The speed of motors can only be within predefined limit
	Executes a PID control loop and issues PWM signals to four motors connected to the four propellers of a quad-copter	Issue the PWM signal within sampling frequency interval (5 Hz in our setup) to ensure the quad-copter is stable	<ul style="list-style-type: none"> Set PWM frequency Set PWM pulse duration (four, one for each of the propellers) 	Parameter corruption attack [17]: modify PID control coefficients and send incorrect PWM pulse to the front right motor	Check PID control coefficients (i.e., pulse duration values) before issuing PWM signals to the motos
	The robot arm performs the following operations in a sequence: pick an object (close it claws), move the arm to destination position, drop the object (open claws) and reset the arm back to initial position	Complete movement of the object before arrival of the next object; inter-arrival duration of the objects was set at 250 ms	Set rotation angle for each of the four servos	Synchronization attack [143]: sends incorrect <code>angle</code> value to the servo channel and prevents the arm from resetting back to its initial position	Check the consistency of each (<code>channel</code> , <code>angle</code>) pair (i.e., the <code>angel</code> value for a given servo <code>channel</code> can not be more than the designer provided bounds) before issuing pulses to the servo motors
	The pump pushes certain amount of fluid and then pulls the trigger to reset the syringe to its initial position	Perform the push/pull operations within designer specified time limit (set at 300 ms)	<ul style="list-style-type: none"> Set motor rotation frequency Drive the motor forward/backward to push the fluids and reset the motor position 	Bolus tampering attack [131, 143]: the attacker injects more fluid than the permitted volume	Checks the amount of fluid the motors can pump (i.e., monitor the number of PUSH/PULL events the controller task invokes)

to evaluate security solutions in cyber-physical applications [13, 17, 44, 46, 62, 143]. I now present the evaluation platforms used in our experiments. Table 6.3 summarizes the properties of each of these systems and attack/detection techniques used in my experiments.

- Case-Study #1 (Ground Rover):** My first case-study platform is a ground rover introduced in Section 6.2.3. There are two motors attached to the rover wheels (e.g., actuators in our context). I used an open-source implementation of the rover controller (written in Python) [144] and ported it to C for compatibility with OP-TEE APIs. The rover performed a line-following mission where it moved from a source to a target way-point by following a line. Each instance of the controller task first set the speed of the motor for the wheel movements and then steered (e.g., forward, left or right) based on its position on the line.

Actuation: The rover has four actuation commands: two for setting the speed of both of the wheels

and two for issuing navigation commands to the two motors attached to the wheel. Table 6.3 lists these actuation commands.

Attack and Consequences: I injected a DoS attack [62] that arbitrarily sets a high speed for one of the motors (to destabilize the rover and move it away from the line). This attack can deviate the rover from its way-points (or even crash it) due to the imbalance in the wheel speeds.

Detection: In my setup the `CheckAct()` functions validates as to whether the rover speed is within designer-given predefined thresholds [145] and verifies whether the navigation commands were consistent with the rover position. I detected the DoS attack by checking the bounds on the speed (i.e., 70–100 decimal values [145]) issued by the controller task.

- **Case-Study #2 (Flight Controller):** My second case-study is a flight-controller for quadcopter [146]. The original controller code is developed for Arduino platforms. Since Arduino boards do not support TrustZone, I adapted it to execute on Raspberry Pi and OP-TEE enabled environments. In this setup the controller executes a PID control loop using the Ziegler–Nichols method [147] and sends pulse width modulation (PWM) signals to spin each of four motors (i.e., actuators) connected to the propellers.

Actuation: There are five actuation commands: one for setting the PWM frequency and other four are for sending PWM pulse durations for each of the motors to rotate the copter propellers. The `CheckAct()` functions verified whether the PWM frequency and pulse durations sent to each of the motors were within a certain range (obtained from PID control logic).

Attack and Consequences: For this case study I considered a parameter corruption attack [17] that modifies the control parameters (e.g., the PID control coefficients) at runtime and sends incorrect pulse values to the front-right motors. This attack can suddenly turn off/freeze the propellers. As a result, the copter will instantly fall/crash.

Detection: This attack is detected since I verify the PID parameters and corresponding PWM pulse durations.

- **Case-Study #3 (Robotic Arm):** My next case-study platform is a robotic arm used in manufacturing systems. The movement of the robotic arm is controlled by four servos (actuators in our context). Each servo is connected to a specific “channel” (I/O port) in the Adafruit motor shield. I use an open-source Python-based robot controller [148] and adapted the implementation for my C-based setup.

Actuation: The robot performed an assembly line sequence with the following four actuation operations: `PICK()`, `MOVE()`, `DROP()` and `RESET()` that (i) picks an object from first position, (ii) moves the arm to a final position, (iii) drops the object and, finally, (iv) resets the arm to initial position (to pick up another object). Each operation takes a (`channel`, `angle`) pair that controls the rotation of the corresponding servo to the desired angle (45° in my setup [148]).

Attack and Consequences: Used a synchronization attack [143] that destabilizes the assembly line by preventing the robot from resetting its arm back to the initial position. To demonstrate this, I injected a logic bomb that sets an incorrect `angle` value in the `RESET()` operation (e.g., servo channel 3). This attack can collapse the whole assembly line since the arm is not returned to the

initial position and hence is unable to pick up objects queued in the line.

Detection: `CheckAct()` detects this attack since it asserts that each servo can only move up to a certain designer-provided angle (45°) for each of the operations.

- **Case-Study #4 (Syringe Pump):** My final case-study platform is a syringe/infusion pump [149]. In our experiments I considered a bolus delivery use-case [131, 150] where the syringe pump first pushes a certain amount of fluid (PUSH event) and then pulls the trigger back (PULL event). The syringe movement is controlled by a stepper motor. Since the original implementation is for Arduino platforms (and does not support TrustZone), I modified the codes to make it compatible with Raspberry Pi and its motor driver library.

Actuation: For a given fluid amount, the syringe pump implementation selects the number of steps where the PUSH and PULL events should be called. I considered each of the PUSH/PULL events as actuation requests since they set the direction of motor rotation. In my setup there were seven actuation requests: one for setting the motor rotation frequency and six for PUSH and PULL events (three each). PULL events were called after all three PUSH operations were completed.

Attack and Consequences: I implemented a bolus tampering attack [131, 143] where the adversary injects more fluid than is required (i.e., more than three PUSH events). The attack has serious safety consequences and is a health hazard since it can inject more fluids/medications to the patient body than the permitted amount.

Detection: This attack is detected by `CheckAct()` since it verifies the motor frequency and how many times each of the PUSH/PULL events are called.

Experience and Findings. In my study I compare SCATE against another scheme that checks *all* the actuation commands. I refer to this latter technique as the “*fine-grain*” checking scheme. Note that in fine-grain checking, there are more context switches between normal and secure execution mode since *all* the commands are checked. The goal of my experiments was to *study the trade-offs between security and real-time requirements*. I therefore considered the subset of commands selected for checking were no more than 50% of the total number of commands so that tasks can finish before their timing requirements (see Table 6.3)⁵ (i.e., $K = \lfloor 0.5N \rfloor$) and assumed equal weights for all commands. I note that my implementation is modular and can be easily adjusted with different weight values. Table 6.4 lists the total number of commands (N) and subset of commands (K) for each of evaluation platforms. I now discuss the results of applying these schemes on the case-studies and address the following research questions (RQs):

- **RQ1.** *How quickly an intrusion can be detected by SCATE when compared to the fine-grain scheme?*
- **RQ2.** *What are the performance impacts and runtime overheads of these schemes?*

⁵I also carried out additional experiments to show the effect of varying this parameter (see Section 6.5.3).

Table 6.4: Number of Actuation Commands

Platform	Total Commands (N)	Selected for Vetting ($K = \lfloor 0.5N \rfloor$)
Ground Rover	4	2
Flight Controller	5	2
Robotic Arm	4	2
Syringe Pump	7	3

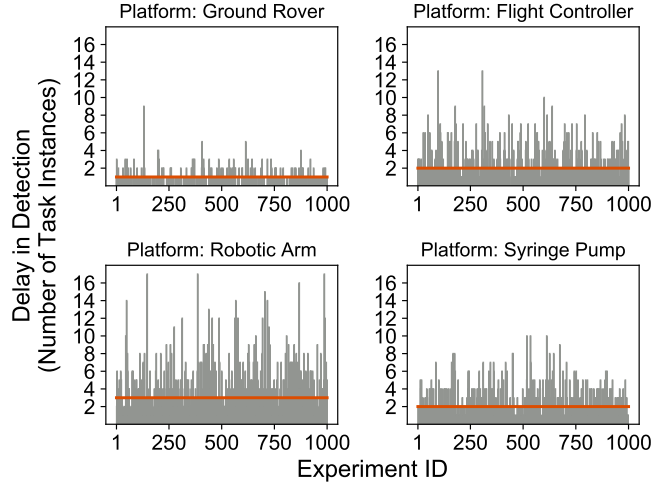


Figure 6.7: Delay in detecting an intrusion (in terms of number of jobs) for SCATE in comparison with the fine-grain scheme. On average (horizontal line), the detection delay is no more than 3 task instances for my case-study platforms.

Security Analysis

In the first set of experiments (Fig. 6.7) I analyze the delay in detecting the attacks: between SCATE and fine-grain scheme. The workflow of my experiments for each of the platforms was as follows: for a given platform and for each of my experiments (x-axis of Fig. 6.7) I triggered the attack at random points in time (i.e., during the execution of the victim task) and measured the time delays (in terms of number of task instances, y-axis in Fig. 6.7) when the corresponding `CheckAct()` function detected the attack. In the fine-grain scheme, the time to detect an attack is upper bounded by the sampling interval (period), T_i (i.e., requires at most one task instance). For a given platform, each point (\hat{x}, \hat{y}) in Fig. 6.7 represents the delay in detecting an attack (when compared to fine-grain checking) at the \hat{x} -th experiment trial is no more than $\hat{y}T_c$ time units (i.e., requires \hat{y} additional task instances) where T_c is the period of the corresponding controller task (see Table 6.3). The horizontal line in the figure shows the mean detection delay. From my experiments, with 1000 individual trials for each of the four platforms, I found that the mean and 99th-percentile detection delay were 1–3 and 3–12 sampling intervals, respectively (refer to Table 6.5 for exact values). I note that this delay in detection results in improve response time and reduced resource

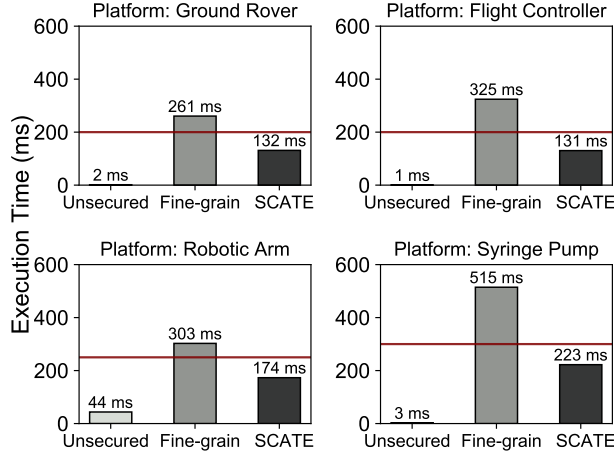


Figure 6.8: Execution time of the main controller task for different schemes. The plots show 99th-percentile values observed from 1000 individual trials. Red horizontal lines represent task deadlines. Fine-grained checking requires more time to compute (due to additional checks and context switching overheads) and often derives the system in unsafe states.

usage (see more in the following experiments) and that could be acceptable for many real-time applications.⁶

Key Findings: *SCATE can provide similar-level of security when compared to the fine-grained checking since on-average it requires only 1–3 additional task instances to detect the attacks.*

Timeliness and Overhead Analysis

The *physical system will remain stable if the controller task can finish execution before its next periodic invocation*. As a reference, I also compare with traces from *vanilla* execution scenario when there is no verification of actuation commands (i.e., tasks are always running in the rich OS). I refer to this vanilla execution as “*unsecured*” since it does not protect the system from any adversarial actions. Figure 6.8 shows the execution time (y-axis) for all three schemes (captured using the Linux `clock_gettime()` function and the `CLOCK_PROCESS_CPUTIME_ID` clock). The vertical line represents the deadlines, i.e., if the response time of the task is above the margin, the task misses its deadline and the physical system will become unstable (and hence unsafe). Both the fine-grained and SCATE increase response times when compared to the unsecured scheme since there is no context switch between Linux and OP-TEE in the latter. I note that the increase in computing resources due to integration of additional security checking/protection techniques (e.g., cryptographic operations, memory isolation, intrusion detection, control flow integrity checks) is an expected side-effect to improve security as observed in prior work [10, 16, 17, 19, 20, 124].

The fine-grained scheme expends more time (than SCATE) since it verifies all N actuation commands, i.e., there are more context switches (from rich OS to secure enclave) and runtime

⁶I discuss this topic further in Chapter 7.

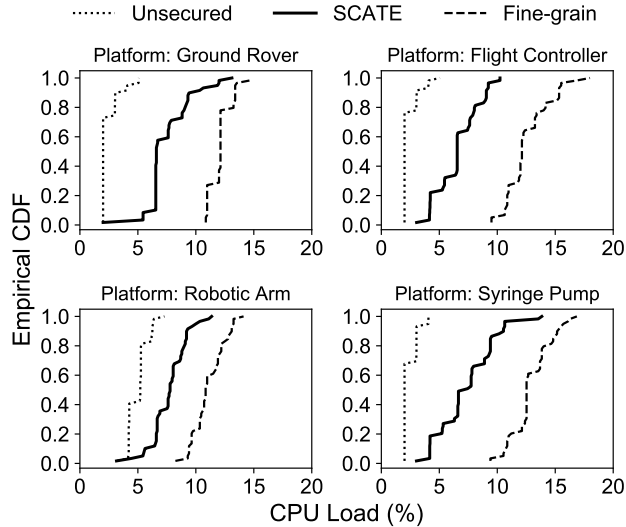


Figure 6.9: Empirical CDF of the CPU load. On average, SCATE uses 30.48%–47.32% less CPU.

checking overheads. As a result, the controller task can easily miss its deadline and drive the system into an unsafe state for all of my case-studies. In contrast, intermittent checking (SCATE) allows the tasks to finish within deadlines. I note that since, by definition, task response times must be less than their periods for stability requirements (Appendix C.1), the controller tasks must be required to execute with a slower frequency (i.e., longer period) if we want to enforce fine-grain checking. For many control systems sampling rates affect the control performance [33]; therefore by selectively checking only a subset of commands, designers can improve task response times and control frequencies while ensuring stability of the physical system.

Key Findings: *Fine-grained checking increases execution time and the controller tasks fail to comply with their timing requirements. SCATE manages to complete execution before the deadline of the tasks.*

In the final set of experiments (Fig. 6.9) I show the resource usage (i.e., CPU load) for all three cases for each platform in our evaluation. For that, I executed the controller tasks independently for 60 seconds and observed the CPU load using `/proc/stat` interface. I report the results from 1000 individual trials. The x-axes of Fig. 6.9 show the CPU load and y-axes show the corresponding cumulative distribution function (CDF). The vertical line shows the average CPU usage. From my experiments I found that SCATE increases CPU usage by 1.5–3.2 times when compared to unsecured scheme – this is expected since vanilla execution does not provide any security guarantees (and there is no context switch overhead). I also note that SCATE reduces CPU load by 30.48%–47.32% when compared to the fine-grain scheme; this could be useful for many applications (say for battery operated systems to improve thermal efficiency).

Key Findings: *In comparison with fine-grained checking, on average, SCATE uses 30.48%–47.32% less CPU for its operation.*

Remarks. My experiments on the four real platforms conclude that SCATE results in slight

Table 6.5: Comparison with Fine-grain Checking: Summary of Findings

Platform	Performance Metrics: SCATE vs Fine-grain			
	Detection Delay (Task Instances)		Execution Time Reduction	CPU Usage Reduction
	Mean	99 th -p	(%)	(%)
Ground Rover	1	3	49.69	37.58
Flight Controller	2	8	59.81	47.32
Robotic Arm	3	12	42.80	30.48
Syringe Pump	2	8	56.81	42.87

Table 6.6: Simulation Parameters

Parameter	Values
Number of processor cores, P	4
Number of tasks, M	[12, 40]
Task periods, T_i	[10, 1000] ms
Number of actuation requests, N_i	{[3, 5], [8, 10]}
Minimum requests verified per job, N_i^{min}	[0.2 N_i]
Verification overhead, C_i^o	10% of C_i

degradation in security (e.g., mean detection delay is at most 3 jobs) while provide significant savings in task response time (i.e., guarantees stability) and resource usage. Table 6.5 summarizes my findings (i.e., delay in detection as well as reduction in execution time and CPU usage) for all four experimental platforms. As we see in the experiments, there is a trade-off between security and real-time requirements. For instance, the fine-grain checking scheme can detect the intrusions faster (i.e., requires at most one additional instances) when compared to SCATE. However it results in the controller tasks taking significantly longer time to finish (i.e., can make the system unstable) and consume more resources (e.g., CPU, battery, memory). By using the mechanisms proposed in SCATE, designers can now measure such trade-offs, evaluate the cost of integrating security and customize the number of security checks for each task that provides the best balance between real-time and security guarantees.

6.5.3 Simulation-based Evaluation

I also developed an open source simulator [116] and conducted experiments with randomly generated workloads for broader design-space exploration. Table 6.6 lists the parameters used in my simulations.

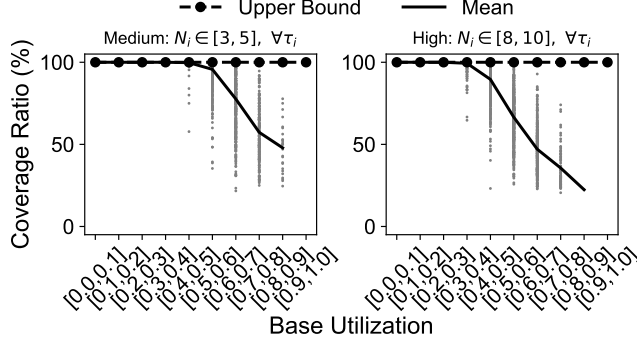


Figure 6.10: Utilization vs coverage ratio. The upper bound represents the fine-grained checking where all the commands are verified. SCATE can provide at least 50% coverage per instance if the system utilization is no more than 70%.

Workload Generation

I considered $P = 4$ cores and each taskset instance contained $[3P, 10P]$ tasks. To generate systems with an even distribution of tasks, I grouped the tasksets by base CPU utilization⁷ from $[(0.01 + 0.1i)P, (0.1 + 0.1i)P]$ where $i \in \mathbb{Z}, 0 \leq i \leq 9$. Each utilization group contained 500 tasksets (i.e., a total $10 \times 500 = 5000$ tasksets were tested). I assumed that the tasks were partitioned using the first-fit strategy [101]. I only considered the feasible tasksets (i.e., the response times are less than deadlines for all tasks) — since tasksets that fail to meet this condition are trivially unschedulable. Task periods were generated according to a log-uniform distribution where each task had periods between $[10, 1000]$ ms. I assumed rate-monotonic priority ordering (i.e., shorter period implies higher priorities) [80]. For a given number of tasks and total system utilization, the utilization of individual tasks were generated using Randfixedsum algorithm [108].

I further assumed that for each task τ_i , the overhead for checking each actuation command (C_i^o) is no more than 10% of task execution time C_i (i.e., $C_i^o = 0.1C_i$). I considered two actuation command request scenarios: (i) medium ($N_i \in [3, 5], \forall \tau_i$) and (ii) high ($N_i \in [8, 10], \forall \tau_i$). I also assumed equal weights for all actuation commands and the minimum number of checking N_i^{min} was at least 20% of total number of requests (i.e., $N_i^{min} = \lceil 0.2N_i \rceil$).

Results

In the following experiments (Fig. 6.10) I study how many commands each task can verify per instance in SCATE. For this, I introduce a metric called “coverage ratio” (CR). The CR metric shows us how many actuation commands (over total number of commands) we can check without violating timing constraints. I define CR as follows: $CR = \frac{1}{|\Gamma|} \sum_{\tau_i \in \Gamma} \frac{K_i}{N_i}$ where $\frac{1}{|\Gamma|} \sum_{\tau_i \in \Gamma} \frac{N_i^{min}}{N_i} \leq CR \leq 1$ $|\Gamma|$ is the total number of tasks and the parameter K_i is obtained from Algorithm 6.1 (see Section 6.4.2). Notice that $CR = 1$ (i.e., upper bound) is the fine-grain

⁷Recall from Section 4.3.1 that the *utilization* of a task is given by the ratio of its execution time to period [80].

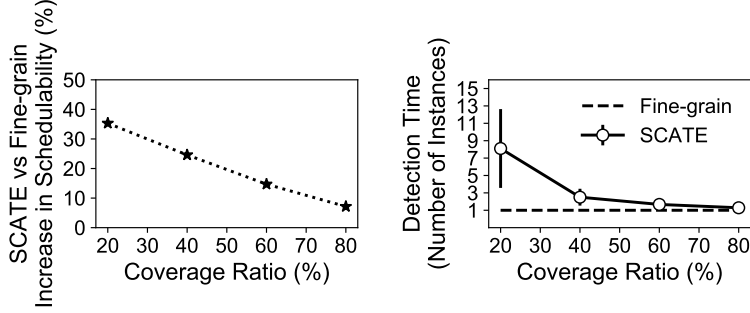


Figure 6.11: Real-time and security trade-offs: low coverage ratio (i.e., when fewer commands are checked in each task instance), while increasing the acceptance ratio (left plot), can lead to increase in detection time (right plot). I set the number of actuation commands at $N_i = 5$.

checking case since it verifies all the commands. Let me now define base-utilization of a taskset (i.e., total utilization without any actuation checking) as $\frac{U}{P}$ where $U = \sum_{\tau_i \in \Gamma} \frac{C_i}{T_i}$, C_i is the task execution time and P_i is the period. The x-axes of Fig. 6.10 show the base-utilization and y-axes show coverage ratio for both medium (top plot) and high (bottom plot) actuation scenarios. From my experiments I find that *SCATE provides similar level of security when compared to fine-grain scheme* if the total utilization is no more than 60% and 40% for medium and high actuation request scenarios, respectively. I note that while fine-grain checking can provide better coverage, this upper bound is otherwise unattainable (specially for high utilization cases) since the all the tasks may not meet their timing requirements. This is seen in my additional experiments (Fig. C.1). In contrast, SCATE can provide at least 50% coverage even in high utilization cases (e.g., $\frac{U}{P} \leq 0.7$), when fine-grained checking fails and makes system infeasible.

Key Findings: *The performance of SCATE is identical to the fine-grain scheme for low-to-medium system utilizations (i.e., able to check all actuation commands). For higher utilizations, SCATE manages to check more than 50% and 25% of the total actuation commands in each task instances for the medium and high scenarios, respectively.*

In the last set of experiments (Fig 6.11) I show the trade-off between real-time and security guarantees. For this, I use the “schedulability” metric — a useful mathematical tool developed by the real-time community to analyze whether all activities of a given system can meet their timing constraints even in the worst-case behavior of the system [23]. A given taskset is considered as *schedulable* if *all* the tasks in the taskset meet their timing requirements (i.e., response time is less than or equal to deadline). If the tasks are not schedulable, the system will be in unsafe state (and should not be deployed). The x-axes of Fig. 6.11 show coverage ratio. The y-axis in the left figure shows the increase in schedulability in SCATE when compared to fine-grain scheme while the right figure shows the detection time (in terms of number of task instances) for both of the schemes. As I mentioned earlier, in the fine-grain scheme, the detection time for a known attack is upper bounded by the period of the task (i.e., requires at most one additional instances when compared to the insecure base-case). As we see from the figure, there is a trade-off between real-

time and security requirements: lower coverage ratio increases the schedulability (since there are lower checking overheads) but increases the detection times. This is because if coverage ratio is low, only a few commands are selected for checking during each instances and a vulnerable/compromised command will only be verified infrequently; thus resulting longer detection times. I also carried out additional experiments (Appendix C.3) to study the trade-offs of integrating TEE-based SCATE mechanism in existing real-time platforms. My results (see Fig. C.1 in Appendix C.3) show that there is a cost of integrating security (since it reduces the number of tasks that meets their timing requirements).

Key Findings: *If we perform fewer checks in each task instances, we can accommodate more tasks in the system (i.e., result in higher acceptance ratio). However, this may result in delayed detections (e.g., on average, requires eight additional task instances) since not all the actuation commands are frequently checked.*

Summary. My experiments reveal interesting trade-offs between real-time and security requirements. Fine-grain checking — while providing better security guarantees (i.e., lower detection time) — can negatively affect schedulability and, hence, safety and integrity of the system. SCATE, in contrast, provides better schedulability guarantees (especially for high utilization cases) but may be result in slower detection times. By using my approach, designers of the systems can now customize their platforms and selectively verify actuation commands based on application requirements.

6.6 CONCLUSION

In this chapter I present a framework, SCATE, to enhance the security and safety of the time-critical cyber-physical systems. I use a combination of trusted hardware and the intrinsic real-time nature of such systems and propose techniques to selectively verify a subset of commands that provides a trade-offs between real-time and security guarantees. I believe that my technique can be incorporated into multiple cyber-physical application domains such as avionics, automobiles, industrial control systems, medical devices, unmanned and autonomous vehicles.

The techniques presented in Chapters 4–5 support a part of my dissertation hypothesis (Section 1.1) by presenting software-based time-aware security integration techniques for both, single core (CONTEGO) and multicore (HYDRA/HYDRA-C) platforms. Chapter 6 further bolsters my claims and demonstrates the efficacy of a hardware/software-based co-design approach (SCATE) to integrate security in single/multicore-based real-time CPS. Hence, the combined frameworks developed in Chapters 4–6 affirm my dissertation hypothesis.

CHAPTER 7: DISCUSSION

My frameworks presented in Chapters 4–5 are not focused towards any specific security mechanisms and allow designers to integrate any given technique based on application requirements. Depending on the actual operation of the security tasks, a particular (class of) attack may or may not be detectable. For instance CONTEGO, HYDRA and HYDRA-C may not detect a zero-day exploit for some security tasks.

The underlying detection algorithms in security tasks could raise false positive errors that may cause CONTEGO to switch modes unnecessarily. Again, a clever adversary may remain undetected and provide a fake indication of malicious activity. This may cause CONTEGO to frequently switch modes thus reducing performance and availability. Although CONTEGO *guarantees that the system will remain schedulable* (and hence safe) even with mode changes (refer to Section 4.5.3), running of security tasks in the ACTIVE mode could impose additional overheads (i.e., increased load as we have seen in Fig. 4.5) that designers of the system may want to avoid. The false-positive/false-negative errors can be mitigated by carefully designing the detection algorithms based on application requirements. Further, I argue that forced mode changes would require an adversary to intrude into the system and *remain undetected for a long time*. In practice that could be *difficult* and *unlikely* in the presence of several intrusion detection tasks.

The security mechanism presented in Chapters 4–5 will collapse if the adversary can compromise *all* the security tasks. To do so, the adversary would need to intrude into the system, remain undetected and monitor the schedule [63] (to override the security tasks) *over a long period of time*. While compromising all the security tasks could be *difficult* in practice, it nevertheless would be worthwhile to check the integrity of the monitoring mechanisms, thus further improving the security posture of the systems. This is an interesting research problem by itself and I investigate this in Chapter 6. In particular, SCATE executes security checks (that verify actuation commands) inside a trusted enclave (ARM TrustZone) and ensures that the protection mechanisms can not be tampered.

Note that checking rules (i.e., *State* \rightarrow *Action* matching) used in SCATE are generally derived from system requirements/specifications and SCATE is compatible with existing techniques [122, 123]. My current implementation aims to block malicious commands. Other strategies could involve the raising of alarms and/or sending out buffered (or even predetermined) alternate commands. I intend to incorporate these features in future work. Chapter 6 assumes the existence of a “perfect” checking module given by the system engineers (i.e., an attack is always correctly detected by the `CheckAct()` function). Depending on the actual implementation, `CheckAct()` functions may result in false-positive/false-negative errors. My model can also handle such cases by incorporating the detection inefficiency factors in the calculation of reward/cost metrics. For example, if the detection accuracy of `CheckAct()` is 95%, one way to express reward and cost functions is as follows: $\lambda_{\text{Imperfect}} = (1 - .05)\lambda$ and $\zeta_{\text{Imperfect}} = (1 + .05)\zeta$, respectively.

While SCATE imposes delays in detection (e.g., on average 1–3 task instances when compared to the scheme that checks all the actuation commands), this could retain the safety and normal operations of the plant due to physical inertia. For example, consider a simplified drone example. If an adversary sends false commands and turns off the propellers, the drone’s altitude will not instantaneously drop to zero. Hence, although SCATE may not detect the attack instantly, it can still block spoofed commands and prevent the drone from crashing. I present additional results and discuss this topic further in Appendix C.4.

In this dissertation I consider fixed-priority scheduling scheme for real-time tasks — since this is the scheduling policy used in a majority of the practical systems. Real-time researchers also propose other schemes such as dynamic-priority [80], global [23] and randomized schedules [43, 151]. My proposed schemes can also be adapted to different real-time scheduling techniques by modifying schedulability conditions and response time expressions. Customizing the proposed frameworks for other scheduling policies, however, require further research.

Although I demonstrate my ideas on various realistic real-time platforms, the lack of real-time benchmarks is one of the major challenges in evaluating real-time cyber-physical security solutions. This is partly because of the diversity of such applications and software/hardware platforms as well as their hardware-dependent nature. In addition, critical cyber-physical applications are rarely open-sourced for safety/security/proprietary reasons. As a result, existing real-time security research is mainly evaluated by using simulations [10, 11, 19, 20] and/or limited case studies [12–15, 18, 39, 40, 62, 83, 152–154].

CHAPTER 8: CONCLUSION AND FUTURE WORK

Modern real-time embedded systems have evolved in a complex manner due to autonomous systems and cloud-like transparent infrastructure. They are also increasingly facing serious security problems. There is a need for a multi-layered, systematic, engineering approach to secure such critical systems. In this dissertation I develop frameworks to integrate security into legacy real-time cyber-physical platforms. I start with a hypothesis that there exist security monitoring mechanisms (such as those check filesystems integrity or verify actuation commands) that can be integrated into real-time CPS using timing-aware techniques. I validate my hypothesis by developing solutions that can integrate independent, periodic security monitoring tasks by imposing scheduler-level constraints — for both, single core (Chapter 4) and multicore platforms (Chapter 5) — without compromising timing/safety guarantees of existing tasks. I further propose a hardware/software co-design approach (Chapter 6). The framework proposed in Chapter 6 prevents attacks that falsify actuation commands by (a) leveraging hardware-assisted security extensions (to ensure integrity of security checks) and (b) applying scheduling and game theory-based optimization techniques (to comply with timing/safety requirements). I demonstrate the efficacy of my integration mechanisms in practical cyber-physical platforms and analyze the design trade-offs — both, from security and real-time perspectives. I believe that solutions developed in this work will help the designers to characterize security of systems. It is my intent that this dissertation work will guide future research efforts and ultimately improve the security of this field.

Correctness of the Dissertation Hypothesis. The aim of this dissertation is to show that a given set of designer-provided security monitoring techniques can be adapted for real-time CPS without compromising timing/safety requirements of existing tasks. Recall from Section 1.1 my dissertation hypothesis states that *it is possible to integrate security into real-time cyber-physical systems by a careful (task/scheduler-level) analysis of, and co-design with, system constraints, viz. software, hardware and timing requirements.* My dissertation work has provided the means to analyze each of the constructs mentioned in the hypothesis. Chapters 4–5 impose scheduler-level constraints to integrate security monitoring tasks. I use intrusion detection mechanisms as security tasks and demonstrate the efficacy of my integration techniques using (a) a custom intrusion detection task (Chapter 5) as well as (b) off-the-shelf integrity checking tools such as Tripwire [60] and Bro [69] (Chapter 4–Chapter 5). I further combine hardware-assisted security features (ARM TrustZone [25]) with scheduling/optimization techniques and develop solutions that prevent falsification of actuation commands while retaining real-time guarantees (Chapter 6).

The techniques developed in this work show that it is possible to integrate security in real-time CPS by imposing time-aware, scheduler-level constraints (Chapters 4–5). I further demonstrate the feasibility of hardware/software-based co-design approaches to integrate security in cyber-physical applications without compromising real-time requirements (Chapter 6). Therefore the conjugated techniques presented in Chapters 4–6 hold my dissertation hypothesis.

Table 8.1: Recommended Security Integration Techniques for Various Scenarios

Constraints		Recommended Techniques
Security Checks	CPS Platform	
Independent of real-time tasks	Does not support trusted executions	CONTEGO (Single core) HYDRA-C (Multicore)
Require application specific checks	Supports trusted executions	SCATE (Single core and Multicore)

Lessons Learned and Recommendations. The techniques proposed in this dissertation analyze various real-time vs. security trade-offs. For instance, the ACTIVE mode in CONTEGO can detect intrusions faster but it requires more resources and delays execution of other low-priority tasks. Likewise, we can ensure better responsive by migrating the security tasks to empty cores at runtime (HYDRA-C) but it increases number of context switches. Selective checks in SCATE result in improved timing and QoS guarantees but comes with a cost of delayed detection.

The frameworks presented in Chapter 5 does not require low-level modifications in the system kernel. Therefore it is possible to deploy continuous security monitoring techniques (e.g., HYDRA-C) using scheduler interfaces provided by the real-time kernels (e.g., RT_PREEMPT [100]). I note that publicly available real-time Linux schedulers do not support adaptive mode switching such as those proposed in Chapter 4. One way to provide adaptive switching is to develop userspace plugins (e.g., by using the Linux /proc interface [155, Ch. 6]) that can interact with the scheduler and switch the execution mode at runtime depending on application requirements. I further note that at present OP-TEE [127] is the only open source, well-documented, active project for developing TrustZone-enabled applications in Linux. Therefore it is required to use the interfaces provided by the OP-TEE developers to integrate off-the-shelf TEE-based checking mechanisms in Linux-based real-time applications.

Based on the techniques developed in this dissertation, I have the following recommendations (listed in Table 8.1) to integrate security in real-time CPS.

- If (a) the security monitoring mechanisms are independent of real-time tasks and (b) the underlying CPS platform does not support trusted enclaves, then CONTEGO and HYDRA-C are the suitable frameworks for securing single and multicore-based real-time platforms, respectively.
- If (a) the security checks depend on the actions of the real-time tasks and (b) they require tamper-proof execution, designers can adapt techniques similar to that presented in SCATE to integrate security in their target platform.

8.1 FUTURE DIRECTIONS

I now highlight possible directions to extend the solutions proposed in this dissertation for a comprehensive security-aware real-time framework.

Non-preemptive Execution. My frameworks allow security tasks to be preempted (by other higher-priority real-time or security tasks) at any point of time. There exist cases when some of the security tasks may need to be executed *without preemption*. For instance, consider a security task that scans the process table and has been preempted in the middle of its operation. An adversary may corrupt the process table entry that has already been scanned before the next scheduling point of the security task. When the security tasks are rescheduled, it will start scanning from its last known state and may not be able to detect the changes in a timely manner. When security tasks need to perform special non-preemptive operation, the priority of the task can be increased to a priority that is strictly higher than all of the real-time tasks. I note that the cost of non-preemption (by means of priority inversion) will compromise the timing constraints of some (or all) of the real-time tasks. Hence, schedulability analysis needs to consider this. Besides, The scheduling policy should identify which real-time or security tasks can be dropped (or perhaps reschedule to other cores) to provide better trade-off between real-time system performance and defense against security vulnerabilities.

Dependency and Reactive Security. The solutions proposed in this dissertation work in a proactive manner. Another direction is to design security integration techniques that *react*, based on anomalous behavior. Hence the executions of security checks require to follow certain *precedence constraints*. For instance, consider a security task checks runtime of real-time tasks. Because of intrusions (or perhaps due to other system artifacts) the monitored task is not behaving as expected. Therefore the security task may perform additional actions to identify the root cause of the problem (e.g., it may check the list of system calls, to see if any undesired calls are executed). In such cases we may not independently execute the security tasks in parallel into multiple cores. One way to support such a feature is to consider the *dependency* between security checks (e.g., checking of system calls depends on runtime behavior of monitored task).

Integration of Cryptographic Primitives. The game theory-based formulation used in Chapter 6 can be extended to other real-time security use-cases. For instance, consider a distributed CPS where real-time nodes periodically exchange messages that need to be encrypted/authenticated (say to prevent man-in-the-middle attacks) [10, 11, 39]. While longer key sizes can provide better encryption, it requires more time to perform crypto operations (hence less number of messages can be secured and/or tasks can miss deadlines). By using game formulations similar to those in Chapter 6, designers of systems can use different (perhaps smaller) key sizes for different messages (to reduce overhead) that provides maximal security from an external observer’s point of view while guaranteeing timing requirements.

Performance Metrics. In this work I use *time-to-detect an intrusion* as a performance metric to observe how well the security checks can perform desired monitoring and detection. While time-to-detect is a useful metric, it is hard to quantify in a comprehensive way as it depends on a number of factors (such as the efficacy of monitoring tasks and the kind of intrusion) and is a lagging metric. I believe that *identifying and designing better security metrics* is an important and challenging research problem.

Response and Recovery Mechanisms. A key reason for detecting attacks early is to provide enough information to system operators so that they can respond to and recover from attacks. Systems with real-time requirements often use autonomous, decision making algorithms for controlling elements in the physical world and there is a need for automatic recovery (on the detection of an attack). The techniques proposed in this work do not consider the after-effects of an intrusion. We need further studies to design *autonomous attack detection, isolation and response algorithms* for safety-critical real-time embedded systems.

APPENDIX A: CONTEGO – SUPPLEMENTARY MATERIALS

A.1 LINEAR LOWER-BOUND SUPPLY FUNCTION AND SCHEDULABILITY CONSTRAINTS

For the security server with unknown capacity Q and replenishment period P , I derive the lower (upper) bound of Q (P) that makes security tasks running under server schedulable by using periodic server model introduced in literature [84, 85, 156]. The key idea from previous work is that a task τ_i can be schedulable if minimum supply for the server can match the maximum workload generated by τ_i and $hp(\tau_i)$ during a time interval t . If the server task τ_S is scheduled by a fixed-priority scheme, the minimum supply of the server is delivered to the security tasks when its $(k - 1)$ -th execution has just finished with minimum interference from the high-priority real-time tasks $\tau_j \in \Gamma_R$. Then, the subsequent executions of k -th release are maximally delayed by the higher-priority real-time tasks. For this minimum supply, we can parameterize the *linear lower-bound supply function* $\mathbf{lsbf}_S(t)$ with the period and execution time of higher-priority real-time tasks.

The worst-case response time of the server is the longest time from the server being replenished to its capacity being exhausted with the maximum interference from the high-priority real-time tasks, given that there are security tasks ready to use all of the server’s available capacity. In order to calculate exact response time of the server, I use the formula introduced in existing work [157]. Using this exact method, we can calculate the maximum possible preemption on the server from the higher-priority real-time tasks for a certain length of window and add up the server’s capacity. The calculation is repeated iteratively by increasing the window size until the window size exceeds the server’s relative replenishment period (in this case the system is determined to be unschedulable) or until the window size is stable. Then, the window size is the *busy period* [158] and let me denote it as w_S .

The worst-case release pattern of server occurs when τ_S and $hp(\tau_S)$ is released simultaneously. The worst-case busy period w_S is the maximum time duration that the server can take to execute full capacity Q when it is released simultaneously with the higher-priority real-time tasks, $hp(\tau_S)$ at the k -th release. By using the traditional exact analysis [157] the worst-case busy period can be obtained as:

$$w_S^{k+1} = Q + \sum_{\tau_h \in hp(\tau_S)} \left\lceil \frac{w_S^k}{T_h} \right\rceil \cdot C_h \quad (\text{A.1})$$

where $w_S^0 = Q$ and $w_S = w_S^{k+1} = w_S^k$ when it converges for some k .

Therefore, the worst-case delay at the k -th release and thereafter can be represented as:

$$\Delta_S = \sum_{\tau_h \in hp(\tau_S)} \left\lceil \frac{w_S}{T_h} \right\rceil \cdot C_h. \quad (\text{A.2})$$

However, such iterative methods are only amenable to brute-force approach. This is because, the ceiling function with unknown value (e.g., the busy period) can not be in our formulation. Thus I take a different approach by approximating Δ_S . During a time interval of P , the maximum workload generated by the server and higher-priority real-time tasks can be represented by:

$$w_S = Q + \sum_{\tau_h \in hp(\tau_S)} \left\lceil \frac{P}{T_h} \right\rceil \cdot C_h. \quad (\text{A.3})$$

Thus using Eq. (A.3), we can avoid the iterative calculation by assuming the number of invocation of higher-priority real-time tasks during P , not during the exact busy period of the server. Since $\lceil y \rceil \leq y + 1$, I linearize w_S by removing the ceiling function and represent Eq. (A.3) as:

$$w_S = Q + \sum_{\tau_h \in hp(\tau_S)} \left(\frac{P}{T_h} + 1 \right) \cdot C_h. \quad (\text{A.4})$$

Therefore, the worst-case linear lower-bound supply function of the security server during a time interval t is given by [156]:

$$\mathbf{lsbf}_S(t) = \frac{Q}{P} [t - (P - Q) - \Delta_S] \quad (\text{A.5})$$

where

$$\Delta_S = \sum_{\tau_h \in hp(\tau_S)} \left(\frac{P}{T_h} + 1 \right) \cdot C_h. \quad (\text{A.6})$$

Let $\Gamma_S^{(\cdot)}$ represents the corresponding security tasksets in the representative mode (i.e., PASSIVE or ACTIVE). In order to derive the minimum capacity that guarantees to schedule $\tau_i \in \Gamma_S^{(\cdot)}$, let me consider the situation when τ_i barely meets its deadline at $t = D_i$ with the worst-case interference from high-priority security tasks, $hp(\tau_i) \in \Gamma_S^{(\cdot)}$. Let me now define the *critical instant* of the security tasks, i.e., the worst-case response time of τ_i when τ_i and $hp(\tau_i)$ are released simultaneously at the end of server's $(k - 1)$ -th execution and suffer worst-case preemptions from k -th release and thereafter [84]. Let me further denote I_i as the worst-case workload generated by the τ_i and $hp(\tau_i)$ from critical instant to deadline of τ_i where I_i is given by:

$$I_i = C_i + \sum_{\tau_h \in hp(\tau_i)} \left\lceil \frac{D_i}{T_h} \right\rceil \cdot C_h. \quad (\text{A.7})$$

In order to ensure the schedulability of the security task τ_i , the minimum supply delivered by the server has to be greater than or equal to the worst-case workload during the time interval D_i , i.e.,

$$\mathbf{lsbf}_S(D_i) \geq I_i \quad \forall \tau_i \in \Gamma_S \quad (\text{A.8})$$

where $\mathbf{lsbf}_S(\cdot)$ is given by Eq. (A.5). Therefore, the constraints on the server supply bound to

ensure schedulability of the security task τ_i can be expressed as:

$$\frac{Q}{P} [D_i - (P - Q) - \Delta_S] \geq I_i. \quad (\text{A.9})$$

It is worth noting that Eq. (A.8) is only a sufficient and not necessary condition. The security task τ_i can be schedulable if and only if there exists a time instance $t \leq D_i$ such that the inequality in Eq. (A.8) holds. I use the sufficient condition in Eq. (A.8) because the presence of time in the necessary condition makes the proposed optimization framework inapplicable to the problem under consideration. Despite the fact that this bound may not be exact and may incur approximation error in the supply function, as I show in Section 4.7, it enables us to integrate security tasks without violating real-time constraints.

A.2 SOLUTION TO THE OPTIMIZATION PROBLEMS

The ACTIVE and PASSIVE modes parameter selection formulations given in Section 4.5 are constrained nonlinear optimization problems and not very straightforward to solve. Therefore I reformulate the optimization problems as a *geometric program* (GP) [86]. A non-linear optimization problem can be solved by GP if the problem is formulated as follows:

$$\min_{\mathbf{x}} f_0(\mathbf{x}) \quad (\text{A.10})$$

$$\text{Subject to: } f_i(\mathbf{x}) \leq 1 \quad i = 1, \dots, z_p \quad (\text{A.11})$$

$$g_i(\mathbf{x}) = 1 \quad i = 1, \dots, z_m \quad (\text{A.12})$$

where $\mathbf{x} = [x_1, x_2, \dots, x_z]^T$ denotes the vector of z optimization variables. The functions $f_0(\mathbf{x}), f_1(\mathbf{x}), \dots, f_{z_p}(\mathbf{x})$ are *posynomial* and $g_1(\mathbf{x}), \dots, g_{z_m}(\mathbf{x})$ are *monomial* functions, respectively. A function $g_i(\mathbf{x})$ is monomial if it can be expressed as:

$$g_i(\mathbf{x}) = c_i \prod_{l=1}^{L_i} x_l^{a_l} \quad (\text{A.13})$$

where $c_i \in \mathbb{R}^+$ and $a_l \in \mathbb{R}$. Note that the coefficient c_i must be non-negative but the exponents a_l can be any real number including fractional and negative.

A posynomial function is the sum of the monomials, and thus can be represented as:

$$f_i(\mathbf{x}) = \sum_{l=1}^{L_i} c_l x_1^{a_{1l}} x_2^{a_{2l}} \dots x_z^{a_{zl}} \quad (\text{A.14})$$

where $c_l \in \mathbb{R}^+$ and $a_{jl} \in \mathbb{R}$. We can maximize a non-zero posynomial objective function by minimizing its inverse.

Period Selection using Geometric Programming Formulation

Observation A.1. *The fundamental measures, i.e., $\sum_{\tau_i \in \Gamma_S^{(\cdot)}} \omega_i \frac{T_i^{des}}{T_i} = \sum_{\tau_i \in \Gamma_S^{(\cdot)}} \omega_i T_i^{des} T_i^{-1}$ and $\sum_{\tau_i \in \Gamma_S^{(\cdot)}} \frac{C_i}{T_i} = \sum_{\tau_i \in \Gamma_S^{(\cdot)}} C_i T_i^{-1}$ in the period adaptation problem are posynomials where $\Gamma_S^{(\cdot)}$ represents the corresponding security tasksets in the representative mode (i.e., PASSIVE or ACTIVE).*

This directly follows from the observation that all the coefficients are non-negative and the variables (e.g., periods) are always positive. Besides, I am only summing up positive terms and therefore the terms are closed under addition. Since the requirement for posynomials is that it need to be closed under addition, the above terms are posynomials.

An interesting property of posynomials and monomials is that, if $f(\cdot)$ is a posynomial and $g(\cdot)$ is a monomial, the ratio $\frac{f(\cdot)}{g(\cdot)}$ will become a posynomial. Since $\frac{f(\cdot)}{g(\cdot)}$ is a posynomial, this allows us to express the constraint $f(\cdot) < g(\cdot)$ as follows: $\frac{f(\cdot)}{g(\cdot)} \leq 1$. For instance, we can easily handle the constraint of the form $f(\cdot) \leq \alpha$ where $f(\cdot)$ is a posynomial and $\alpha > 0$. We can refer $\hat{f}(\cdot)$ is an *inverse posynomial* if $\frac{1}{\hat{f}(\cdot)}$ is a posynomial. Besides, we can maximize a non-zero posynomial objective function by minimizing its inverse.

Based on the above description, I reformulate the maximization problem as a standard GP minimization problem in either mode as follows:

$$\min_{\mathbf{T}^{(\cdot)}} \sum_{\tau_i \in \Gamma_S^{(\cdot)}} \omega_i^{-1} (T_i^{des})^{-1} T_i \quad (\text{A.15})$$

$$\text{Subject to: } \left(\sum_{\tau_i \in \Gamma_S^{(\cdot)}} C_i T_i^{-1} \right) \cdot \left(n \left[\left(\frac{3 - \frac{Q^{(\cdot)}}{P^{(\cdot)}}}{3 - 2 \frac{Q^{(\cdot)}}{P^{(\cdot)}}} \right)^{\frac{1}{n}} - 1 \right] \right)^{-1} \leq 1 \quad (\text{A.16})$$

$$(3P^{(\cdot)} - 2Q^{(\cdot)}) T_i^{-1} \leq 1 \quad \forall \tau_i \in \Gamma_S^{(\cdot)} \quad (\text{A.17})$$

$$T_i^{des} T_i^{-1} \leq 1 \quad \forall \tau_i \in \Gamma_S^{(\cdot)} \quad (\text{A.18})$$

$$(T_i^{max})^{-1} T_i \leq 1 \quad \forall \tau_i \in \Gamma_S^{(\cdot)} \quad (\text{A.19})$$

where for any symbol $y^{(\cdot)}$ represents the corresponding variable in the representative mode (e.g., PASSIVE or ACTIVE).

The above GP formulation is not a convex optimization problem since the posynomials are not convex functions [86]. However, it can be converted into a convex optimization problem using logarithmic transformations (i.e., based on a logarithmic change of variables, as well as a logarithmic transformation of objective and constraint functions). In particular, by using logarithmic transformations (i.e., representing $\tilde{T}_i = \log T_i$ and hence $T_i = e^{\tilde{T}_i}$, and replacing inequality constraints of the form $f_i(\cdot) \leq 1$ with $\log f_i(\cdot) \leq 0$), I convert the above formulation into a convex optimization problem. This convex optimization reformulation is solvable in polynomial time by using standard algorithms, such as the *interior-point* method [87, Ch. 11].

Selection of Server Parameters

Theorem A.1. *The objective functions (i.e., the ratio between server capacity and period) and the server schedulability constraints (i.e., Eq. (4.15) and Eq. (4.20)) can be expressed in posynomial form.*

Proof. The proof follows by rearranging the terms in posynomial form and transform the objective functions into minimization expression. Let me rearrange the objective function (i.e., ratio between capacity and period) as QP^{-1} which is clearly a posynomial. The objective functions in **P4.4** and **P4.5** can be rewritten as a standard GP minimization problem as follows:

$$\min_{Q^{(\cdot)}, P^{(\cdot)}} Q^{(\cdot)-1} P^{(\cdot)} \quad (\text{A.20})$$

where $Q^{(\cdot)}$ and $P^{(\cdot)}$ represent the server parameters in PASSIVE and ACTIVE modes. Note that Eq. (A.20) is also in posynomial form. Let me now rearrange server schedulability constraints as follows:

$$(Q^{(\cdot)} + \Delta_{S^{(\cdot)}})P^{(\cdot)-1} \leq 1 \quad (\text{A.21})$$

where

$$\Delta_{S^{(\cdot)}} = \sum_{\tau_h \in hp(\tau_S^{(\cdot)})} (P^{(\cdot)} + T_h) \cdot T_h^{-1} \cdot C_h. \quad (\text{A.22})$$

Since the optimization variables (e.g., capacity and replenishment period) are always positive, using the similar argument presented in Observation A.1, I can assert that Eq. (A.21) is a posynomial constraint. QED.

In order to express the schedulability constraints for the security tasks (i.e., Eqs. (4.16) and (4.21)) as as posynomial form the me rearrange the equations as follows:

$$\frac{P^{(\cdot)}(Q^{(\cdot)} + I_i) + \Delta_{S^{(\cdot)}}Q^{(\cdot)}}{Q^{(\cdot)}(Q^{(\cdot)} + T_i^*)} \leq 1 \quad \forall \tau_i \in \Gamma_S^{(\cdot)}. \quad (\text{A.23})$$

Recall that, if we want to represent the constraint of the form $\frac{f^{(\cdot)}}{g^{(\cdot)}} \leq 1$ the denominator must be monomial. However, the inequality in Eq. (A.23) does not conform to a posynomial form due to the posynomial term in the denominator, i.e., $Q^{(\cdot)}(Q^{(\cdot)} + T_i^*) = (Q^{(\cdot)})^2 + QT_i^*$. The following theorem illustrates how the constraints on server bound can be represented in posynomial form.

Theorem A.2. *The server bound constraints can be formulated as the following posynomial form:*

$$\left[P^{(\cdot)}(Q^{(\cdot)} + I_i) + \Delta_{S^{(\cdot)}}Q^{(\cdot)} \right] \cdot \left[Q^{(\cdot)} \cdot \hat{g}(Q^{(\cdot)}, T_i^*) \right]^{-1} \leq 1 \quad \forall \tau_i \in \Gamma_S^{(\cdot)}. \quad (\text{A.24})$$

Proof. The theorem is proved by using the geometric mean approximation [159, Ch. 2] of posynomials. Since the denominator in Eq. (A.23) is a posynomial, let me approximate $Q + T_i^*$ with a monomial by the following geometric mean approximation.

Let me denote $Q + T_i^*$ as $g(Q^{(\cdot)}, T_i^*) = u_1(Q^{(\cdot)}) + u_2(T_i^*)$ where $u_1(Q^{(\cdot)}) = Q^{(\cdot)}$ and $u_2(T_i^*) = T_i^*$. We can approximate $g(Q^{(\cdot)}, T_i^*)$ with

$$\hat{g}(Q^{(\cdot)}, T_i^*) = \left[\frac{u_1(Q^{(\cdot)})}{a} \right]^a \cdot \left[\frac{u_2(T_i^*)}{b} \right]^b \quad (\text{A.25})$$

where $a = \frac{u_1(y_0)}{g(y_0, T_i^*)}$, $b = \frac{u_2(T_i^*)}{g(y_0, T_i^*)}$ and $y_0 \in \mathbb{R}^+$ is a constant that satisfies $\hat{g}(y_0, T_i^*) = g(y_0, T_i^*)$. The approximated monomial $\hat{g}(Q^{(\cdot)}, T_i^*)$ can be rewritten as

$$\hat{g}(Q^{(\cdot)}, T_i^*) = \left(\frac{Q}{a} \right)^a \cdot \left(\frac{T_i^*}{b} \right)^b \quad (\text{A.26})$$

where $a = \frac{y_0}{y_0 + T_i^*}$, $b = \frac{T_i^*}{y_0 + T_i^*}$. Using this monomial approximation, I represent Eq. (A.23) as:

$$\frac{P^{(\cdot)}(Q^{(\cdot)} + I_i) + \Delta_{S^{(\cdot)}} Q^{(\cdot)}}{Q^{(\cdot)} \cdot \hat{g}(Q^{(\cdot)}, T_i^*)} \leq 1 \quad \forall \tau_i \in \Gamma_S^{(\cdot)} \quad (\text{A.27})$$

and the proof follows.

QED.

Likewise, the real-time task schedulability constraints for the ACTIVE mode (e.g., Eq. (4.18)) can be represented as follows:

$$\left(C_j + \sum_{\tau_h \in hp_R(\tau_j)} \left\lceil \frac{D_j}{T_h} \right\rceil C_h \right) D_j^{-1} + \left(D_j (P^{ac})^{-1} Q^{ac} + Q^{ac} \right) D_j^{-1} \leq 1, \quad \forall \tau_j \in lp_R(\tau_S^{ac}). \quad (\text{A.28})$$

After the logarithmic transformation (i.e., $\tilde{Q}_i^{(\cdot)} = \log Q_i^{(\cdot)}$, $\tilde{P}_i^{(\cdot)} = \log P_i^{(\cdot)}$ and replacing the inequality constraints $f_i(\cdot) \leq 1$ with $\log f_i(\cdot) \leq 0$), the objective function and the constraints become a standard convex optimization problem that is solvable in polynomial time.

I also present another approach and carry out additional experiments to obtain server parameters by exhaustively searching all possible period and capacity values. My findings are presented in the following section.

A.3 COMPARISON WITH EXACT METHOD

I now compare the GP-based approach with an exhaustive search method¹ based on exact analysis. In this exhaustive search, I assign server replenishment period from 1 to P_{max} with a granularity of δ . For each period, I determine the minimum capacity requirements that makes the tasks schedulable. From the set of feasible period and capacity pair, I take the pair that maximizes the server utilization. Notice that, the minimum server capacity $Q_{min}(\tau_i, P)$ for $\tau_i \in \Gamma_S$

¹Similar search method has also been discussed in literature [67, 84, 156].

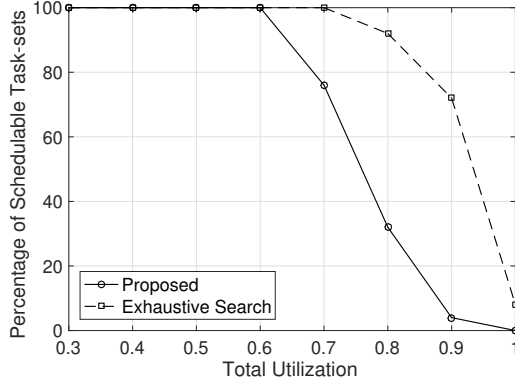


Figure A.1: Normalized percentage of the number of schedulable task-sets. The base-utilization of the real-time tasks were varied from $[0.01 + 0.1 \cdot i, 0.1 + 0.1 \cdot i]$ where $0 \leq i \leq 8, i \in \mathbb{Z}$. The utilizations of the security tasks were generated from $[0.11, 0.20]$. For exhaustive search, I set $P_{max} = 2500$ with search granularity $\delta = 0.5$.

with a given P can be obtained by solving the quadratic inequality in Eq. (A.9), which is given by:

$$Q_{min}(\tau_i, P) = \frac{-(D_i - P - \Delta_S) + \sqrt{(D_i - P - \Delta_S)^2 + 4I_i P}}{2}. \quad (\text{A.29})$$

In the above equation Δ_S is calculated by exact method, i.e.,

$$\Delta_S = \sum_{\tau_h \in hp(\tau_S)} \left\lceil \frac{w_S}{T_h} \right\rceil \cdot C_h \quad (\text{A.30})$$

where w_S is obtained from Eq. (A.1). In order to find the minimum required capacity of the server for a given replenishment period P , I take the maximum of the capacity $Q_{min}(\tau_i, P)$ over all the security tasks $\tau_i \in \Gamma_S$ which is defined as:

$$Q_{min}(P) = \max_{\tau_i \in \Gamma_S} \{Q_{min}(\tau_i, P)\}. \quad (\text{A.31})$$

Hence any Q from $[Q_{min}(P), P]$ such that $Q + \Delta_S \leq P$ will be the feasible capacity (i.e., makes the task-set schedulable).

In Fig. A.1 I compare the number of schedulable task-sets found in the proposed method and exhaustive search. For exhaustive search, I set $P_{max} = 2500$ with granularity $\delta = 0.5$. As we can see from figure, the difference in terms of schedulable task-sets found by exhaustive search compared to GP increases for higher base-utilization. I attribute that due to approximation of supply function in the security server. Recall that, the exhaustive search method calculates minimum capacity of the server by exact analysis of the busy period. In contrast, the proposed method approximates the interference to the server from real-time tasks during the interval of server replenishment period and linearize it by taking the ceiling off. While this approximation error is small for low utilization

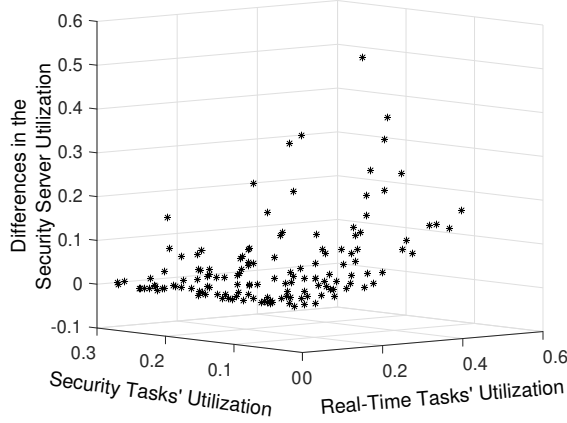


Figure A.2: Exhaustive search vs. GP-based optimization: difference in the server utilization for schedulable tasksets. For each utilization group, I randomly generated 100 task-sets and compared the schedulability of both schemes.

cases, as the base-utilization increases the error accumulates and reduces schedulability. However, still it is possible to accumulate task-sets for higher base-utilization.

The quality of solution (i.e., server utilization) obtained by GP and exhaustive search is illustrated in Fig. A.2. The z-axis in this figure represents the difference in server utilization, i.e., $\left(\frac{Q^{\text{EX}}}{P^{\text{EX}}} - \frac{Q^{\text{GP}}}{P^{\text{GP}}}\right)$ where Q^{EX} and Q^{GP} (P^{EX} and P^{GP}) represent the capacity (replenishment period) obtained from exhaustive search and proposed method, respectively. For low-to-medium utilization cases, the difference is close to zero, which implies the quality of the solution obtained by the GP method is similar to that of obtained by exhaustive search. However, when the utilization is higher the exhaustive search outperforms the proposed method. Again, I attribute this due to the approximation of supply function in the security server.

It is worth noting that the solution obtained by exhaustive search may not be optimal in a sense that the actual replenishment period may appear beyond P_{max} . As we can see from Fig. A.2, for some task-sets the difference is less than zero, i.e., $\frac{Q^{\text{EX}}}{P^{\text{EX}}}$ is lower than $\frac{Q^{\text{GP}}}{P^{\text{GP}}}$. I highlight that the actual search region to find the optimal server parameters for exhaustive search may widely vary based on task-set inputs; and can only be found numerically by trial-and-error. Instead, the proposed method provides a generic approach to analyze the system that is independent of task-set input parameters.

I note that the proposed GP-based approach can solve a given task-set in *seconds*, while the exhaustive search method generally takes few minutes to couple of hours depending on the size of P_{max} and the search granularity δ . Besides, the exhaustive search method is not scalable for task-sets with large number of tasks.

APPENDIX B: HYDRA/HYDRA-C – SUPPLEMENTARY MATERIALS

B.1 SOLUTION TO THE PERIOD SELECTION PROBLEM IN HYDRA

The period adaptation problem given in Section 5.3.2 is a constrained optimization problem and not straightforward to solve. Therefore I reformulate the optimization problems as a GP [86]. Based on the concepts presented in Appendix A.2, we can represent period selection problem as a GP. Let me rearrange the objective function as follows: $\min_{T_s} (T_s^{des})^{-1}$. Likewise period bound constraint in Eq. (5.4) can be represented as $T_s^{des}T_s^{-1} \leq 1$ and $(T_s^{max})^{-1}T_s \leq 1$, respectively. In addition, the schedulability constraint in Eq. (5.6) can be rewritten as: $(C_s + I_s^m)T_s^{-1} \leq 1$ where

$$I_s^m = \sum_{\tau_r \in \Gamma_R} \mathbb{I}_r^m (T_r + T_s) T_r^{-1} C_r + \sum_{\tau_h \in hp_S(\tau_s)} x_h^m (T_h + T_s) T_h^{-1} C_h. \quad (\text{B.1})$$

The above reformulation is not a convex optimization problem since the posynomials are not convex functions [86]. However, by using logarithmic transformations (i.e., representing $\tilde{T}_s = \log T_s$ and hence $T_s = e^{\tilde{T}_s}$, and replacing inequality constraints of the form $f_i(\cdot) \leq 1$ with $\log f_i(\cdot) \leq 0$), we can convert the above formulation into a convex optimization problem that can be solved using standard algorithms, such as the *interior-point* method in polynomial time [87, Ch. 11].

B.2 COMPARING HYDRA WITH OPTIMAL MULTICORE ASSIGNMENT

I now empirically compare HYDRA an “optimal” multicore allocation scheme. The result of an empirical comparison of HYDRA with an optimal solution (i.e., a solution of the formulation described in Section 5.3.1 that finds the variables \mathbf{X} and \mathbf{T}) is presented in Fig. B.1 where I exhaustively searched for all possible combinations for a small setup with $M = 2$ cores and up to $N_S = 6$ security tasks. To find the optimal solution, I examined each of the M^{N_S} possible assignments of security tasks to cores. For each assignment, I then determined the value of the period vector \mathbf{T} that maximized the cumulative tightness by solving a convex optimization problem (see Appendix B.1).

The x-axis of Fig. B.1 represents total system utilization and y-axis is the difference in cumulative tightness (i.e., $\Delta_\eta = \frac{\eta_{\text{OPT}} - \eta_{\text{HYDRA}}}{\eta_{\text{OPT}}} \times 100\%$) for HYDRA and the optimal solution. As shown in the figure, for low to medium utilization cases, HYDRA’s performance is similar to the optimal solution (i.e., the difference is zero). However for higher utilizations performance degrades. This is because HYDRA follows an iterative best-fit strategy to find the periods (and assignment). Hence for higher utilization values the lower priority tasks may not get periods close to the desired values (and the cumulative tightness degrades). As we see from the figure, the degradation (in cumulative tightness) is no more than 22% and that may be acceptable given the exponential computational complexity of finding an optimal solution.

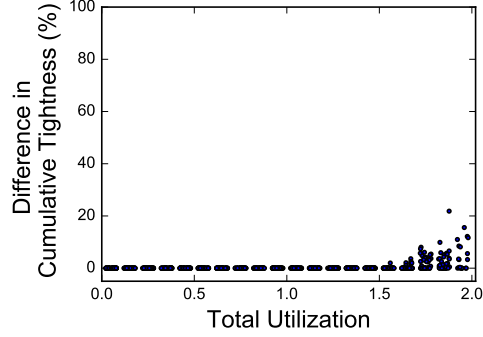


Figure B.1: Comparing HYDRA with optimal solution: I consider $M = 2$ and $N_S \in [2, 6]$ with other parameters similar to that mentioned in Section 5.5.2.

B.3 PROOF OF LEMMA 5.1

Since real-time tasks are partitioned and they have higher priorities than security tasks, the schedule of real-time tasks executed on π_m does not depend on any other task in the system. Now consider any interval $[t, t + x)$ of length x . I show that we can obtain an interval $[t', t' + x)$ where all tasks are released at t' , such that the workload of real-time tasks on π_m is higher in $[t', t' + x)$ compared to $[t, t + x)$.

First step: let t' be the earliest time such that π_m continuously executes real-time tasks in $[t', t)$; if such time does not exist, then let $t' = t$. By definition, π_m does not execute real-time tasks at time $t' - 1$. Also since real-time tasks continuously execute in $[t', t)$, the workload of real-time tasks in $[t', t' + x)$ cannot be smaller than the workload in $[t, t + x)$.

Second step: since π_m is idle at $t' - 1$, no job of real-time tasks on π_m released before t' can contribute to the workload in $[t', t)$. Hence, the workload can be maximized by anticipating the release of each real-time task τ_r so that it corresponds with t' . This concludes the proof.

APPENDIX C: SCATE – SUPPLEMENTARY MATERIALS

C.1 SYSTEM MODEL: REAL-TIME TASK AND SCHEDULING

I consider a system consists with M fixed-priority real-time tasks $\Gamma = \{\tau_i, \dots, \tau_M\}$ running on P identical processor cores $\Pi = \{\pi_1, \dots, \pi_P\}$. In this work, I consider a partitioned fixed-priority preemptive scheduling [23] (a widely supported approach in many commercial and open-source real-time operating systems such as QNX [99], OKL4 [98], real-time Linux [100], etc.) where tasks are statically assigned to the processor cores using a predefined partitioning scheme. The set of tasks running on a given core π_p is denoted by Γ_p and $\Gamma = \cup_{\pi_p \in \Pi} \{\Gamma_p\}$. Each task τ_i is represented by the following tuple:

$$(C_i, T_i, D_i, N_i, N_i^{min}, W_i). \tag{C.1}$$

Each of the above variables represents the following:

- C_i is the constant, upper bound on the computation time, called worst-case execution time (WCET) [79];
- T_i is the minimum inter-arrival time (period), i.e., consecutive jobs of τ_i should be temporally separated by at least T_i time units;
- D_i (usually less than or equal to T_i) is the timing constraint (deadline);
- N_i is the number of actuation requests that τ_i sends out;
- $N_i^{min} \leq N_i$ is a QoS parameter that denotes the minimum number of actuation commands that must be checked; and
- $W_i = \{\omega_i^1, \dots, \omega_i^{N_i}\}$ is a designer-provided weight vector where the weight ω_i^j represents the importance of j -th actuation command over other.

As we see in Section 6.4, the parameters N_i^{min} and W_i help the designers to determine the subset of commands to be selected in each job of the task for checking when not all N_i commands can be checked due to timing constraints. While I represent the above task model as above for ease of presentation, I note that not all the real-time tasks in a given system may invoke actuation requests. For such tasks $\tau_{i'}$ setting $N_{i'} = N_{i'}^{min} = 0$ and ignoring the variable $W_{i'}$ will hold the consistency of the representation.

I consider a discrete time model [160] where the system and task parameters are multiples of a time unit, i.e., an interval starting from time point t_1 and ending at time point t_2 that has a length of $t_2 - t_1$ by $[t_1, t_2)$ or $[t_1, t_2 - 1]$. I also assume that the non-secure system (i.e., when there is no actuation command checking) is “schedulable”, that is, for each task $\tau_i \in \Gamma$, the response time (time between completion and arrival) is less than the deadline of the task.

C.2 FEASIBILITY CONDITIONS

Let N_i be the number of actuation requests generated by τ_i that require vetting and C_i^o is an upper bound of additional computing time due to (a) context switching (from normal execution to secure enclave and returning the context back to normal mode) and (b) perform checking inside the enclave. Then the WCET of τ_i can be represented as follows:

$$C_i^{TEE} = C_i + N_i C_i^o. \quad (\text{C.2})$$

The task τ_i is “schedulable” if its worst-case response time (WCRT), R_i^{TEE} , is less than deadline, i.e., $R_i^{TEE} \leq D_i$. We can calculate an upper bound of R_i^{TEE} using traditional response-time analysis [101] as follows:

$$R_i^{TEE} = C_i^{TEE} + \sum_{\tau_h \in hp(\tau_i, \pi_p)} \left(1 + \frac{D_i}{T_h}\right) C_h^{TEE} \quad (\text{C.3})$$

where $hp(\tau_i, \pi_p) \in \Gamma_p$ denotes the set of tasks that are higher-priority than τ_i running on core π_p . The taskset Γ is referred to as schedulable if all the tasks are schedulable, viz., $R_i^{TEE} \leq D_i, \forall \tau_i \in \Gamma$.

Let $R_i = C_i + \sum_{\tau_h \in hp(\tau_i, \pi_p)} \left(1 + \frac{D_i}{T_h}\right) C_h$ denote the vanilla response time (i.e., when there is no actuation checking). Notice that the task τ_i will miss its deadline if $R_i^{TEE} > D_i$. From Eq. (C.3) we can deduce that τ_i will miss its deadline if the following condition holds: $O_i > D_i - R_i$ where

$$O_i = N_i C_i^o + \sum_{\tau_h \in hp(\tau_i, \pi_p)} \left(1 + \frac{D_i}{T_h}\right) N_i C_h^o \quad (\text{C.4})$$

is the total overhead for checking the actuation commands.

C.3 DESIGN-TIME TESTS FOR INTEGRATING ACTUATION CHECKING IN EXISTING SYSTEMS

I also performed experiments to show the impact of integrating TEE-based actuation checking mechanisms (e.g., SCATE and the fine-grain scheme) in an existing system. For this, I use the “schedulability” metric introduced in Section 6.5.3. To demonstrate the effect of schedulability for a large number of tasksets with different parameters, let me now introduce the notion of *acceptance ratio* that is defined as the number of schedulable tasksets over the total number of generated tasksets (e.g., 500 for a given utilization group in my setup).

In Fig. C.1 I compare the performance of difference schemes in terms of acceptance ratio (y-axis in the figure). The x-axis shows the normalized base-utilization $\frac{U}{P}$ where $U = \sum_{\tau_i \in \Gamma} \frac{C_i}{T_i}$ (i.e., taskset utilization without any actuation checking). As expected, schedulability drops for high utilization cases since less number of tasks meet their deadlines due to increased load. While non-secure execution (i.e., when there is no command verification) results in better schedulability

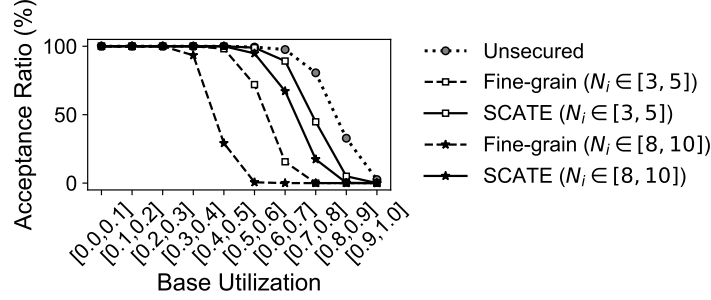


Figure C.1: Impact on schedulability: Fine-grained checking can reduce schedulability significantly (specially if tasks have large number of actuation requests) due to increased validation overheads.

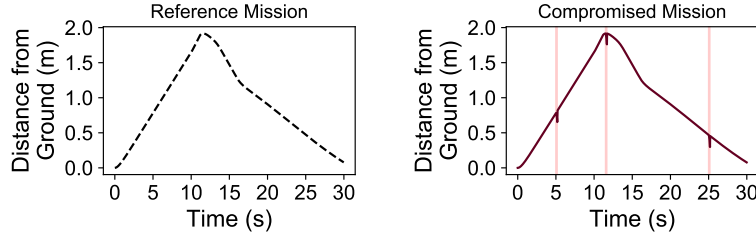


Figure C.2: Effect of physical inertia in cyber-physical applications. The left plot shows the expected altitudes of the drone during mission. The right plot presents the altitudes during attacks. While there exist slight drifts in altitudes before SCATE detects false commands (shaded areas in the right plot), it does not jeopardize the safety (i.e., the drone was above from the ground and did not crash).

due to reduced utilization, it *does not provide any security guarantee*. The fine-grain checking, while providing better security (since it verifies every request), performs poorly in terms of meeting the timing guarantees (essentially keeping the system safe) specifically for highly loaded systems. (i.e., less number of tasks found to be schedulable) due to more validation overheads. In contrast, SCATE provides better schedulability with (slight) QoS/security degradation as we demonstrate in Section 6.5.3. The designers of the systems can use the results presented here to analyze the feasibility of integrating TEE-based checking method in their target platforms.

C.4 IMPACT OF PHYSICAL INERTIA

I now present the impacts of physical inertia to detect attacks in SCATE. For instance, consider a simplified drone example. The baseline safety requirement for the drone is not to crash into the ground during flight. I use existing quad-copter models [161] and simulate the dynamics of the drone for 30 seconds (x-axes in Fig. C.2). In this mission, the drone takes-off from the ground and then lands in the target position. The y-axes in Fig. C.2 represent altitudes of the drone (i.e., distance from the ground) during the mission. The left plot of Fig. C.2 shows the corresponding altitudes over time during the normal quad-copter operation. To demonstrate malicious activity, I injected attacks that sent false commands to turn off the propellers (right plot of Fig. C.2).

In particular, I triggered attacks at the following three instances, viz., *(i)* while the quad-copter was ascending (at 5 sec.), *(ii)* in the peak altitude (at 12 sec.) and *(iii)* during descent (at 25 sec.). The attacks were detected by SCATE within 8 task instances (i.e., 99-th percentile values obtained from the flight controller case-study, see Table 6.5). The vertical lines (light red) in the plot represent time difference when an attack is triggered and when it is detected by SCATE. As the figure illustrates, there is a slight drift in altitude before SCATE detects and blocks false commands. However, this delayed detection does not jeopardize safety constraints (i.e., it does not drop the drone's altitude to zero) and the drone is able to complete the mission without crashing. Hence it is not inconceivable that the detection delays induced by SCATE will be acceptable for many cyber-physical applications.

REFERENCES

- [1] A. Hussain, M. Hannan, A. Mohamed, H. Sanusi, and A. Ariffin, "Vehicle crash analysis for airbag deployment decision," *Int. J. of Auto. Tech.*, vol. 7, no. 2, pp. 179–185, 2006.
- [2] K. Castelli, A. M. A. Zaki, and H. Giberti, "Development of a practical tool for designing multi-robot systems in pick-and-place applications," *MDPI Robotics*, vol. 8, no. 3, 2019.
- [3] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, p. 6, 2011.
- [4] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the ukrainian power grid," *SANS Industrial Control Systems*, 2016.
- [5] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham et al., "Experimental security analysis of a modern automobile," in *IEEE S&P*, 2010, pp. 447–462.
- [6] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno et al., "Comprehensive experimental analyses of automotive attack surfaces," in *USENIX Sec. Symp.*, 2011.
- [7] S. S. Clark and K. Fu, "Recent results in computer security for medical devices," in *MobiHealth*, 2011, pp. 111–118.
- [8] Joon Son and Alves-Foss, "Covert timing channel analysis of rate monotonic real-time scheduling algorithm in MLS systems," in *IEEE Inf. Ass. Wor.*, 2006, pp. 361–368.
- [9] H. Teso, "Aircraft hacking: Practical aero series," in *HITB Sec. Conf.*, 2013.
- [10] T. Xie and X. Qin, "Improving security for periodic tasks in embedded systems through scheduling," *ACM TECS*, vol. 6, no. 3, p. 20, 2007.
- [11] M. Lin, L. Xu, L. T. Yang, X. Qin, N. Zheng, Z. Wu, and M. Qiu, "Static security optimization for real-time systems," *IEEE Trans. on Indust. Info.*, vol. 5, no. 1, pp. 22–37, 2009.
- [12] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo, "S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems," in *ACM HiCoNS*, 2013, pp. 65–74.
- [13] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha, "SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems," in *IEEE RTAS*, 2013, pp. 21–32.
- [14] M.-K. Yoon, S. Mohan, J. Choi, and L. Sha, "Memory heat map: anomaly detection in real-time embedded systems using memory behavior," in *ACM/EDAC/IEEE DAC*, 2015, pp. 1–6.
- [15] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha, "Learning execution contexts from system call distribution for anomaly detection in smart embedded system," in *ACM/IEEE IoTDI*, 2017, pp. 191–196.

- [16] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, “Securing real-time microcontroller systems through customized memory view switching.” in *NDSS*, 2018.
- [17] H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Xinyan, “Detecting attacks against robotic vehicles: A control invariant approach,” in *ACM CCS*, 2018, pp. 801–816.
- [18] F. Abdi, C.-Y. Chen, M. Hasan, S. Liu, S. Mohan, and M. Caccamo, “Preserving physical safety under cyber attacks,” *IEEE IoT J.*, vol. 6, no. 4, pp. 6285–6300, 2018.
- [19] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. B. Bobba, “Real-time systems security through scheduler constraints,” in *Euromicro ECRTS*, 2014, pp. 129–140.
- [20] R. Pellizzoni, N. Paryab, M.-K. Yoon, S. Bak, S. Mohan, and R. B. Bobba, “A generalized model for preventing information leakage in hard real-time systems,” in *IEEE RTAS*, 2015, pp. 271–282.
- [21] D. Lo, M. Ismail, T. Chen, and G. E. Suh, “Slack-aware opportunistic monitoring for real-time systems,” in *IEEE RTAS*, 2014, pp. 203–214.
- [22] F. Abdi, J. Woude, Y. Lu, S. Bak, M. Caccamo, L. Sha, R. Mancuso, and S. Mohan, “On-chip control flow integrity check for real time embedded systems,” in *IEEE CPSNA*, 2013, pp. 26–31.
- [23] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM CSUR*, vol. 43, no. 4, pp. 35:1–35:44, 2011.
- [24] M. Sabt, M. Achemlal, and A. Bouabdallah, “Trusted execution environment: What it is, and what it is not,” in *IEEE Trustcom/BigDataSE/ISPA*, 2015, pp. 57–64.
- [25] S. Pinto and N. Santos, “Demystifying ARM TrustZone: A comprehensive survey,” *ACM CSUR*, vol. 51, no. 6, p. 130, 2019.
- [26] V. Conitzer and T. Sandholm, “Computing the optimal strategy to commit to,” in *ACM EC*, 2006, pp. 82–90.
- [27] A. Burns and R. I. Davis, “A survey of research into mixed criticality systems,” *ACM CSUR*, vol. 50, no. 6, p. 82, 2018.
- [28] S. Vestal, “Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance,” in *IEEE RTSS*, 2007, pp. 239–243.
- [29] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns, “A review of priority assignment in real-time systems,” *Elsevier J. of sys. arch.*, vol. 65, pp. 64–82, 2016.
- [30] A. Gujarati, F. Cerqueira, and B. B. Brandenburg, “Schedulability analysis of the Linux push and pull scheduler with arbitrary processor affinities,” in *Euromicro ECRTS*, 2013, pp. 69–79.
- [31] S. Kato and N. Yamasaki, “Semi-partitioned fixed-priority scheduling on multiprocessors,” in *IEEE RTAS*, 2009, pp. 23–32.

- [32] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, “Partitioned fixed-priority preemptive scheduling for multi-core processors,” in *Euromicro ECRTS*, 2009, pp. 239–248.
- [33] E. Bini and A. Cervin, “Delay-aware period assignment in control systems,” in *IEEE RTSS*, 2008, pp. 291–300.
- [34] A. Aminifar, P. Eles, Z. Peng, and A. Cervin, “Control-quality driven design of cyber-physical systems with robustness guarantees,” in *DATE*, 2013, pp. 1093–1098.
- [35] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, “Period optimization for hard real-time distributed automotive systems,” in *ACM DAC*, 2007, pp. 278–283.
- [36] K. Tindell, H. Hanssmom, and A. J. Wellings, “Analysing real-time communications: Controller area network (CAN).” in *IEEE RTSS*, 1994, pp. 259–263.
- [37] C.-Y. Chen, M. Hasan, and S. Mohan, “Securing real-time Internet-of-things,” *MDPI Sensors*, vol. 18, no. 12, 2018.
- [38] H. Chai, G. Zhang, J. Zhou, J. Sun, L. Huang, and T. Wang, “A short review of security-aware techniques in real-time embedded systems,” *J. of Cir., Sys. and Comp.*, vol. 28, no. 02, 2019.
- [39] V. Lesi, I. Jovanov, and M. Pajic, “Network scheduling for secure cyber-physical systems,” in *IEEE RTSS*, 2017, pp. 45–55.
- [40] V. Lesi, I. Jovanov, and M. Pajic, “Security-aware scheduling of embedded control tasks,” *ACM TECS*, vol. 16, pp. 188:1–188:21, 2017.
- [41] C. Bao and A. Srivastava, “A secure algorithm for task scheduling against side-channel attacks,” in *ACM TrustED*, 2014, pp. 3–12.
- [42] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. B. Bobba, “Integrating security constraints into fixed priority real-time schedulers,” *RTS Journal*, vol. 52, no. 5, pp. 644–674, 2016.
- [43] 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 *IEEE RTAS*, 2016, pp. 1–12.
- [44] 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 *ACM/IEEE ICCPS*, 2018, pp. 10–21.
- [45] R. Liu and M. Srivastava, “PROTC: PROTeCting drone’s peripherals through ARM trustzone,” in *ACM DroNet*, 2017, pp. 1–6.
- [46] P. Guo, H. Kim, N. Virani, J. Xu, M. Zhu, and P. Liu, “RoboADS: Anomaly detection against sensor and actuator misbehaviors in mobile robots,” in *IEEE/IFIP DSN*, 2018, pp. 574–585.
- [47] F. Fei, Z. Tu, R. Yu, T. Kim, X. Zhang, D. Xu, and X. Deng, “Cross-layer retrofitting of UAVs against cyber-physical attacks,” in *IEEE ICRA*, 2018, pp. 550–557.

- [48] S. Moothedath, D. Sahabandu, J. Allen, A. Clark, L. Bushnell, W. Lee, and R. Poovendran, “A game-theoretic approach for dynamic information flow tracking to detect multi-stage advanced persistent threats,” *IEEE TACON*, 2020.
- [49] J. Chen and Q. Zhu, “A game-theoretic framework for resilient and distributed generation control of renewable energies in microgrids,” *IEEE Trans. on Smart Grid*, vol. 8, no. 1, pp. 285–295, 2016.
- [50] G. Yang, R. Poovendran, and J. P. Hespanha, “Adaptive learning in two-player stackelberg games with continuous action sets,” in *IEEE CDC*, 2019, pp. 6905–6911.
- [51] S. Rass, A. Alshawish, M. A. Abid, S. Schauer, Q. Zhu, and H. De Meer, “Physical intrusion games – optimizing surveillance by simulation and game theory,” *IEEE Access*, vol. 5, pp. 8394–8407, 2017.
- [52] A. Humayed, J. Lin, F. Li, and B. Luo, “Cyber-physical systems security – A survey,” *IEEE IoT J.*, vol. 4, no. 6, pp. 1802–1831, 2017.
- [53] Y. Yang, L. Wu, G. Yin, L. Li, and H. Zhao, “A survey on security and privacy issues in Internet-of-Things,” *IEEE IoT J.*, vol. 4, no. 5, pp. 1250–1258, 2017.
- [54] M. Ammar, G. Russello, and B. Crispo, “Internet of Things: A survey on the security of IoT frameworks,” *Elsevier J. of Inf. Sec. & App.*, vol. 38, pp. 8–27, 2018.
- [55] W. Li, H. Chen, and H. Chen, “Research on ARM TrustZone,” *ACM GetMobile*, vol. 22, no. 3, pp. 17–22, 2019.
- [56] L. Sha, “Using simplicity to control complexity,” *IEEE Software*, vol. 18, no. 4, pp. 20–28, 2001.
- [57] X. Wang, N. Hovakimyan, and L. Sha, “L1Simplex: Fault-tolerant control of cyber-physical systems,” in *ACM/IEEE ICCPS*, 2013, pp. 41–50.
- [58] X. Zhang, J. Zhan, W. Jiang, Y. Ma, and K. Jiang, “Design optimization of security-sensitive mixed-criticality real-time embedded systems,” in *IEEE ReTiMiCS*, 2013.
- [59] K. Jiang, P. Eles, and Z. Peng, “Optimization of secure embedded systems with dynamic task sets,” in *DATE*, 2013, pp. 1765–1770.
- [60] “Tripwire,” <https://github.com/Tripwire/tripwire-open-source>.
- [61] R. Mahfouzi, A. Aminifar, S. Samii, M. Payer, P. Eles, and Z. Peng, “Butterfly attack: Adversarial manipulation of temporal properties of cyber-physical systems,” in *IEEE RTSS*, 2019, pp. 93–106.
- [62] M. Hasan and S. Mohan, “Protecting actuators in safety critical IoT systems from control spoofing attacks,” in *ACM IoT S&P*, 2019, pp. 8–14.
- [63] C.-Y. Chen, S. Mohan, R. Pellizzoni, R. B. Bobba, and N. Kiyavash, “A novel side-channel in real-time schedulers,” in *IEEE RTAS*, 2019, pp. 90–102.
- [64] S. Liu, N. Guan, D. Ji, W. Liu, X. Liu, and W. Yi, “Leaking your engine speed by spectrum analysis of real-time scheduling sequences,” *J. of Sys. Arch.*, vol. 97, pp. 455–466, 2019.

- [65] M. Bechtel and H. Yun, “Denial-of-service attacks on shared cache in multicore: Analysis and prevention,” in *IEEE RTAS*, 2019, pp. 357–367.
- [66] F. Loi, A. Sivanathan, H. H. Gharakheili, A. Radford, and V. Sivaraman, “Systematically evaluating security and privacy for consumer IoT devices,” in *ACM IoTS&P*, 2017, pp. 1–6.
- [67] R. Davis and A. Burns, “An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems,” in *IEEE RTNS*, 2008.
- [68] “AIDE,” <http://aide.sourceforge.net/>.
- [69] “The Bro network security monitor,” <https://www.bro.org>.
- [70] M. Roesch, “Snort - lightweight intrusion detection for networks,” in *USENIX Conf. on Sys. Admin.*, 1999, pp. 229–238.
- [71] L. L. Woo, M. Zwolinski, and B. Halak, “Early detection of system-level anomalous behaviour using hardware performance counters,” in *DATE*, 2018, pp. 485–490.
- [72] V. M. Weaver, “Linux perf_event features and overhead,” in *IEEE FastPath*, 2013.
- [73] “OProfile,” <http://oprofile.sourceforge.net/>.
- [74] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM CSUR*, vol. 41, no. 3, p. 15, 2009.
- [75] J. Song, G. Fry, C. Wu, and G. Parmer, “CAML: Machine learning-based predictable system-level anomaly detection,” in *IEEE CERTS*, 2016, pp. 12–18.
- [76] S. Mohan, “Worst-case execution time analysis of security policies for deeply embedded real-time systems,” *ACM SIGBED Review*, vol. 5, no. 1, p. 8, 2008.
- [77] S. K. Baruah, A. Burns, and R. I. Davis, “Response-time analysis for mixed criticality systems,” in *IEEE RTSS*, 2011, pp. 34–43.
- [78] A. K. Mok, “Fundamental design problems of distributed systems for the hard-real-time environment,” Massachusetts Institute of Technology, Tech. Rep., 1983.
- [79] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra et al., “The worst-case execution-time problem – overview of methods and survey of tools,” *ACM TECS*, vol. 7, no. 3, p. 36, 2008.
- [80] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *JACM*, vol. 20, no. 1, pp. 46–61, 1973.
- [81] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, “Applying new scheduling theory to static priority pre-emptive scheduling,” *SE Journal*, vol. 8, no. 5, pp. 284–292, 1993.
- [82] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein, “Analysis of hierarchical fixed-priority scheduling,” in *Euromicro ECRTS*, 2002, pp. 173–181.
- [83] M. Hasan, S. Mohan, R. B. Bobba, and R. Pellizzoni, “Exploring opportunistic execution for integrating security into legacy hard real-time systems,” in *IEEE RTSS*, 2016, pp. 123–134.

- [84] M.-K. Yoon, J.-E. Kim, R. Bradford, and L. Sha, “Holistic design parameter optimization of multiple periodic resources in hierarchical scheduling,” in *DATE*, 2013, pp. 1313–1318.
- [85] I. Shin and I. Lee, “Periodic resource model for compositional real-time guarantees,” in *IEEE RTSS*, 2003, pp. 2–13.
- [86] S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi, “A tutorial on geometric programming,” *Opt. & Eng.*, vol. 8, no. 1, pp. 67–127, 2007.
- [87] S. Boyd and L. Vandenberghe, *Convex optimization*, 2004.
- [88] “CONTEGO implementation,” <https://github.com/mnwrhsn/contego>.
- [89] E. Bini and G. C. Buttazzo, “Measuring the performance of schedulability tests,” *RTS Journal*, vol. 30, no. 1-2, pp. 129–154, 2005.
- [90] A. Mutapcic, K. Koh, S. Kim, L. Vandenberghe, and S. Boyd, “GGPLAB: a simple Matlab toolbox for geometric programming,” 2006. [Online]. Available: <https://stanford.edu/~boyd/ggplab/>
- [91] “BeagleBone Black,” <https://beagleboard.org/black>.
- [92] “Xenomai – real-time framework for Linux,” <https://xenomai.org>.
- [93] “UAV control codes,” <https://github.com/Khan-drone/flight-control>.
- [94] “FreeRTOS,” <http://www.freertos.org>.
- [95] “FTP brute-force attack trace,” <https://github.com/bro/bro/blob/master/testing/btest/Traces/ftp/bruteforce.pcap>.
- [96] *Ethical Hacking and Countermeasures: Secure Network Operating Systems and Infrastructures*, 2nd ed. EC-Council, 2017.
- [97] “Linux ARM shellcode,” <https://www.exploit-db.com/exploits/21253/>.
- [98] G. Heiser and B. Leslie, “The OKL4 microvisor: Convergence point of microkernels and hypervisors,” in *ACM APSys*, 2010, pp. 19–24.
- [99] F. Kolnick, “The QNX 4 real-time operating system,” *Basis Comp. Sys. Inc.*, 1998.
- [100] L. Fu and R. Schwebel, “Real-time Linux wiki,” https://rt.wiki.kernel.org/index.php/rt_preempt_howto, [Online].
- [101] J. Chen, “Partitioned multiprocessor fixed-priority scheduling of sporadic real-time tasks,” in *Euromicro ECRTS*, 2016, pp. 251–261.
- [102] S. Baruah and N. Fisher, “The partitioned multiprocessor scheduling of sporadic task systems,” in *IEEE RTSS*, 2005.
- [103] N. Guan, M. Stigge, W. Yi, and G. Yu, “New response time bounds for fixed priority multiprocessor scheduling,” in *IEEE RTSS*, 2009, pp. 387–397.
- [104] D. E. Knuth, *The art of computer programming: sorting and searching*, 1997, vol. 3.

- [105] “HYDRA-C implementation,” <https://github.com/mnwrhsn/multicore-continuous-security-monitoring>.
- [106] “Raspberry Pi,” <https://tinyurl.com/rpi3modelb>.
- [107] “Linux rootkit,” <https://github.com/crudbug/simple-rootkit>.
- [108] P. Emberson, R. Stafford, and R. I. Davis, “Techniques for the synthesis of multiprocessor tasksets,” in *WATERS*, 2010, pp. 6–11.
- [109] Y. Sun and M. Di Natale, “Assessing the pessimism of current multicore global fixed-priority schedulability analysis,” in *ACM SAC*, 2018, pp. 575–583.
- [110] T. Roughgarden, “Algorithmic game theory,” *Comm. of the ACM*, vol. 53, no. 7, pp. 78–86, 2010.
- [111] J. Westling, “Future of the Internet of things in mission critical applications,” 2016.
- [112] E. Simmon, K.-S. Kim, E. Subrahmanian, R. Lee, F. De Vault, Y. Murakami, K. Zettsu, and R. D. Sriram, *A vision of cyber-physical cloud computing for smart networked systems*. US Dept. of Commerce, NIST, 2013.
- [113] V. Costan and S. Devadas, “Intel SGX Explained,” *IACR Crypt. ePrint Arch.*, no. 086, pp. 1–118, 2016.
- [114] J. C. Harsanyi and R. Selten, “A generalized nash solution for two-person bargaining games with incomplete information,” *INFORMS Man. Sci.*, vol. 18, no. 5-part-2, pp. 80–106, 1972.
- [115] P. Paruchuri, J. P. Pearce, M. Tambe, F. Ordonez, and S. Kraus, “An efficient heuristic approach for security against multiple adversaries,” in *IFAAMAS AAMAS*, 2007, pp. 1–8.
- [116] “SCATE implementation,” https://github.com/mnwrhsn/scate_implementation.
- [117] “Dexter Industries Sensors,” <https://github.com/DexterInd/DI.Sensors>.
- [118] “I²C manual,” Philips Semiconductors, 2003. [Online]. Available: <https://tinyurl.com/i2c-manual>
- [119] Y. A. Korilis, A. A. Lazar, and A. Orda, “Achieving network optima using Stackelberg routing strategies,” *IEEE/ACM TON*, vol. 5, no. 1, pp. 161–173, 1997.
- [120] J. Cardinal, M. Labbé, S. Langerman, and B. Palop, “Pricing of geometric transportation networks,” in *CCCG*, 2005, pp. 92–96.
- [121] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu, “Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-things,” in *ACM HotNets*, 2015, pp. 1–7.
- [122] S. Adepur and A. Mathur, “From design to invariants: Detecting attacks on cyber physical systems,” in *IEEE QRS-C*, 2017, pp. 533–540.
- [123] R. Berthier and W. H. Sanders, “Specification-based intrusion detection for advanced metering infrastructures,” in *IEEE PRDC*. IEEE, 2011, pp. 184–193.

- [124] A. Mukherjee, T. Mishra, T. Chantem, N. Fisher, and R. Gerdes, “Optimized trusted execution for hard real-time applications on cots processors,” in *ACM RTNS*, 2019, pp. 50–60.
- [125] J. Amacher and V. Schiavoni, “On the performance of arm trustzone,” in *IFIP DAIS*, 2019, pp. 133–151.
- [126] Y. Liu, K. An, and E. Tilevich, “RT-trust: Automated refactoring for trusted execution under real-time constraints,” in *ACM GPCE*, 2018, pp. 175–187.
- [127] “Open Portable Trusted Execution Environment,” <https://www.op-tee.org/>.
- [128] “ARM Fixed Virtual Platforms,” <https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms>.
- [129] L. Vandenberghe, “The CVXOPT linear and quadratic cone program solvers,” 2010. [Online]. Available: <http://cvxopt.org/documentation/coneprog.pdf>
- [130] “The Python-MIP package,” <https://www.python-mip.com/>.
- [131] L. Cheng, K. Tian, and D. D. Yao, “Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks,” in *ACM ACSAC*, 2017, pp. 315–326.
- [132] R. Liu and M. Srivastava, “VirtSense: Virtualize Sensing through ARM TrustZone on Internet-of-Things,” in *ACM SysTEX*, 2018, pp. 2–7.
- [133] T. Liu, A. Hojjati, A. Bates, and K. Nahrstedt, “Alidrone: Enabling trustworthy proof-of-alibi for commercial drone compliance,” in *IEEE ICDCS*, 2018, pp. 841–852.
- [134] “Adafruit motor shield for Raspberry Pi,” <https://learn.adafruit.com/adafruit-motor-shield>.
- [135] “Adafruit DC and stepper motor driver source code,” https://github.com/threebrooks/AdafruitStepperMotorHAT_CPP.
- [136] “PCA9685 I2C PWM driver,” <https://github.com/TeraHz/PCA9685>.
- [137] “GlobalPlatform TEE client API specifications,” <https://globalplatform.org/specs-library/tee-client-api-specification/>.
- [138] K. Martin, “Tutorial: COIN-OR: Software for the OR community,” *INFORMS Interfaces*, vol. 40, no. 6, pp. 465–476, 2010.
- [139] A. Lipowski and D. Lipowska, “Roulette-wheel selection via stochastic acceptance,” *Elsevier Physica A*, vol. 391, no. 6, pp. 2193–2196, 2012.
- [140] “Z1FFER open source hardware random number generator,” <http://www.openrandom.org>.
- [141] “Open hardware random number generator,” <https://onerng.info>.
- [142] F. M. Tabrizi and K. Pattabiraman, “Flexible intrusion detection systems for memory-constrained embedded systems,” in *IEEE EDCC*, 2015, pp. 1–12.
- [143] M. R. Aliabadi, A. A. Kamath, J. Gascon-Samson, and K. Pattabiraman, “Artinali: dynamic invariant detection for cyber-physical system security,” in *ACM ESEC/FSE*, 2017, pp. 349–361.

- [144] “Raspberry Pi rover,” <https://github.com/Veilkrand/simplePiRover>.
- [145] “GoPiGo,” <https://github.com/DexterInd/GoPiGo>.
- [146] “Drone controller,” <https://github.com/lobodol/drone-flight-controller>.
- [147] K. J. Åström and T. Häggglund, “Revisiting the Ziegler–Nichols step response method for PID control,” *Elsevier J. of Proc. Con.*, vol. 14, no. 6, pp. 635–650, 2004.
- [148] “Robot arm control,” <https://github.com/tutRPi/6DOF-Robot-Arm>.
- [149] “C-FLAT implementation,” <https://github.com/control-flow-attestation/c-flat>.
- [150] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, “C-FLAT: control-flow attestation for embedded systems software,” in *ACM CCS*, 2016, pp. 743–754.
- [151] H. Baek and C. M. Kang, “Scheduling randomization protocol to improve schedule entropy for multiprocessor real-time systems,” *MDPI Symmetry*, vol. 12, no. 5, p. 753, 2020.
- [152] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, “Contego: An adaptive framework for integrating security tasks in real-time systems,” in *EuroMicro ECRTS*, 2017, pp. 23:1–23:22.
- [153] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, “A design-space exploration for allocating security tasks in multicore real-time systems,” in *DATE*, 2018, pp. 225–230.
- [154] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, “Period adaptation for continuous security monitoring in multicore systems,” in *DATE*, 2020.
- [155] “Linux Kernel Workbook,” <https://lkw.readthedocs.io/>.
- [156] L. Almeida and P. Pedreiras, “Scheduling within temporal partitions: response-time analysis and server design,” in *ACM EMSOFT*, 2004, pp. 95–103.
- [157] M. Joseph and P. Pandya, “Finding response times in a real-time system,” *The Comp. J.*, vol. 29, no. 5, pp. 390–395, 1986.
- [158] J. P. Lehoczky, “Fixed priority scheduling of periodic task sets with arbitrary deadlines,” in *IEEE RTSS*, 1990, pp. 201–209.
- [159] M. Chiang, *Geometric programming for communication systems*, 2005.
- [160] D. Iovic, “Handling sporadic tasks in real-time systems: Combined offline and online approach,” Tech. Rep., June 2001.
- [161] T. Luukkonen, “Modelling and control of quadcopter,” School of Science, Aalto University, Tech. Rep., 2011.