INTEGRATING SECURITY INTO LEGACY REAL-TIME CYBER-PHYSICAL SYSTEMS

BY

MONOWAR HASAN

THESIS PROPOSAL

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:

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

Abstract

Modern real-time (RT) cyber-physical systems (CPS) are increasingly facing security threats than the past. A simplistic straightforward integration of security mechanisms might not be able to guarantee the *safety* and predictability of such systems. In this proposal, I focus on *integrating security mechanisms as a first-class principle* for the design of real-time systems (RTS). My research (both the accomplished and proposed work) addresses the following key challenges:

- Develop techniques and design-time evaluation frameworks to integrate security into RTS, especially for the existing (*viz.,* legacy) ones, where modification of the system architecture/parameters is not an option.
- Define a *metric* to measure the effectiveness of such integration.
- Study system behaviors (such as actuation outputs) and develop hardware/software-based protection mechanisms – for both, RT and general-purpose CPS.

Studies from my previous work show that how engineers can *(a)* integrate security into *single and multicore* RT platforms and *(b)* evaluate the *cost* of such integration. The key of my work is to strive for balance between ensuring the security of the systems and preserving the timing constraints (*i.e.,* deadlines) of existing RT tasks. Based on the my prior work, I further propose a security integration framework for general-purpose CPS that will monitor program behavior (*e.g.,* actuation outputs) and provide runtime protection (from tampering).

Table of Contents

Chapter 1

Introduction

Embedded real-time systems (RTS) are pervasive and are found in everyday use, *e.g.,* automobiles, industrial and process control systems as well as in critical infrastructures (such as electrical grids, oil and gas infrastructures, *etc.*). RTS are also essential part of avionics and used in manned and unmanned aerial vehicles (UAVs) such as airplanes, drones, spacecraft, *etc.* Given their application in safety critical domains, successful attacks or failures in RTS can have catastrophic consequences for the environment and/or to the human safety [1,2].

Attack demonstrations by researchers on automobiles [2,3] and medical devices [4] have shown that systems composed of RTS might be vulnerable to cyber attacks. A number of high-profile attacks on real systems (*e.g.,* Stuxnet [5], BlackEnergy [6]) have shown that the threat is real. Traditional safety and fault-tolerance mechanisms used in RTS were designed to counter random or accidental faults and failures and cannot deal with intentional cyber attacks orchestrated by an intelligent and capable adversary. Further, the drive towards *(i)* use of standardized protocols and commodity-off-the-shelf (COTS) components for interoperability reduced infrastructure and maintenance costs, and *(ii)* smart and connected systems (*e.g.,* smart and connected communities, smart grids, smart or cyber manufacturing, smart transportation, *etc.*) is reducing any protection against cyber attacks that the use of proprietary components and being air-gapped (*i.e.,* unconnected to external systems) might have provided.

Recognizing this emerging threat and urgent need, there has been a lot of focus on securing RTS in the recent past including integrating communication confidentiality [7,8], communication integrity [9,10] and monitoring and detection mechanisms [11–15]. The focus of my research is on integrating or retrofitting security mechanisms into those *legacy* RTS. A legacy (*i.e.,* existing) RTS is one where modification or perturbation of existing real-time tasks' parameters (such as run-times, period and task execution order) is not always feasible. It is not straightforward to retrofit legacy RTS with security mechanisms that

were developed for more general purpose computing scenarios. This is because, when integrating any security mechanisms into RTS, the designers need to ensure that they do not impact the real-time (RT) functionalities in any significant way while at the same time provide the necessary level of security. For instance, in legacy systems the schedule of the existing RT tasks *cannot easily be changed to accommodate security mechanisms*.

In my research I therefore address the following problem:

> *Any security mechanisms have to co-exist with the existing tasks in the system and have to operate without impacting the timing and safety constraints of the control logic.*
>
> Challenge: **how do we integrate and then characterize the effects of security in legacy real-time cyber-physical systems for both single and multicore platforms?**

The key objective of my research is to develop security integration framework for legacy cyber-physical systems (CPS), especially those with RT requirements, and study performance trade-offs. The key questions I want to answer are:

1. How to *integrate security tasks* in RTS without perturbing timing/safety constrains of RT tasks and also *increase the effectiveness of monitoring* and detection mechanisms with better responsiveness? **[Chapter 2]**

2. How to extend the security integration framework for *general-purpose CPS* (where application tasks are not necessarily periodic and may have softer RT requirements) and develop techniques to monitor/verify specific program behavior (such as *actuation commands* invoked by the tasks)? **[Chapter 3]**

Chapter 2

Integrating Security into Legacy RTS

2.1 Overview

In this research I aim to improve the security posture of RTS through integration of 'security tasks' (*e.g.*, tasks that are specific for intrusion monitoring and detection tasks purposes) into an existing fixed-priority system while ensuring that the existing RT 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) [16]. In Table 2.1 I present some examples of security tasks that can be integrated into legacy systems (however, this is by no stretch meant to be an exhaustive list). In my experiments I considered intrusion detection as a monitoring mechanism to demonstrate the feasibility of my approach – the ideas proposed here though apply more broadly to other security mechanisms.

Table 2.1: Example of Security Tasks

Security Task	**Approach/Tools**
File-system checking	Tripwire [17], AIDE [18], *etc.*
Network packet monitoring	Bro [19], Snort [20], *etc.*
Hardware event monitoring	Statistical analysis based checks [21] using performance monitors (*e.g.*, perf [22], OProfile [23], *etc.*)
Application specific checking	Behavior-based detection [12, 24]

2.2 Considerations for Integrating Security Mechanisms

I consider incorporating security mechanisms by implementing them as separate *periodic tasks*. In order to provide the best protection, these security tasks may need to be executed quite often. This now creates

an apparent tension between security requirements (*e.g.,* having enough cycles for effective monitoring detection) and the timing and safety requirements. For example, *how often and how long should a monitoring and detection task run* to be effective but *not interfere* with control or other safety-critical RT tasks? 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. In contrast, if the security tasks are executed very frequently then it may impact the schedulability of the RT tasks – herein lies an important *trade-off between monitoring frequency and schedulability* of RT/security tasks. Specifically, this brings up the challenge of determining the *right periods* (*viz.,* minimum inter-execution time) for the security tasks [25]. For instance, some critical security tasks may be required to execute more frequently than others. However, if the period is too short (*e.g.,* the security task repeats too often) then it will use too much of the processor time and eventually lower the overall system utilization (and consequently performance of the underlying control system). As a result, the security mechanism itself might prove to be a hindrance to the system and reduce the overall functionality or worse, safety. In contrast, if the period is too long, the security task may not always detect violations since attacks could be launched between two instances of the security task. In addition, if the security tasks always execute with lowest priority (since this will not impact the operation of RT tasks), they suffer more interference (*i.e.,* preemption from high-priority RT tasks) and the consequent longer detection time (due to poor response time) will make the security tasks less effective.

The use of multicore platforms in safety-critical RTS is increasingly becoming common since they provide higher performance and better energy efficiency [26]. 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) dedicate a core* to all the security tasks; or *(ii) spread them out* (in conjunction with the RT 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* for all of the above cases?

As we shall see in Section 2.4, my prior work [27–30] addresses the above challenges and find right periods for the security tasks.

2.3 System and Security Model

2.3.1 Real-Time Task Model

In this proposal, I consider a set of N_R RT tasks $\Gamma_R = \{\tau_1, \tau_2, \cdots, \tau_{N_R}\}$, scheduled on platform with M identical cores $\mathcal{M} = \{\pi_1, \pi_2, \cdots, \pi_M\}$ where $M = 1$ implies a single core system. Each RT 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 (*e.g.,* period) and D_r is the relative deadline. We assume constrained deadlines for RT tasks (*e.g.,* $D_r \leq T_r$) and the task priorities are assigned according to rate-monotonic (RM) [31] order (*i.e.,* shorter period implies higher priority). For multicore platforms (*i.e.,* $M > 1$), RT tasks are scheduled using partitioned fixed-priority preemptive scheme [26, 32]. I further assume that the RT tasks are *schedulable*, *i.e.,* the worst-case response time (WCRT) is less than deadline.

2.3.2 Security Tasks

As I mentioned earlier, I propose to integrate security mechanisms as independent periodic tasks. Let us denote N_S security tasks by the set $\Gamma_S = \{\tau_1, \tau_2, \cdots, \tau_{N_S}\}$. I propose to 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 τ_s for effective security monitoring and/or intrusion detection), 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. I also assume that the priorities of 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.

One fundamental problem in integrating security tasks is to determine *which* security tasks will be running *when* [25]. Although any period T_s within the range $T_s^{des} \leq T_s \leq T_s^{max}$ would be acceptable, the actual period T_s, however, is not known a priori. One may wonder why designers cannot assign the desired period (*e.g.,* $T_s = T_s^{des}$) so that the security tasks can always execute with the desired frequency F_s^{des}. However, without careful priority assignment and schedulability analysis, if we set $T_s = T_s^{des}$, $\forall \tau_s$ this may cause high degree of interference to low-priority (RT/security) tasks and results in an unschedulable system. Therefore my goal is to find *suitable periods for the security tasks* without violating RT constraints.

2.3.3 Threat Model

My focus in this research is on integrating security mechanisms (abstracted as security tasks) into an existing (legacy) RTS without impacting its RT functionalities. I assume that an adversary may destabilize the system by leveraging (known/zero-day) 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, *etc.*) to launch further attacks (say denial of service). While there exists mechanisms (such as Simplex [33,34]) 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 system. The addition of such tasks may necessitate changing the schedule or increasing the WCET of RT tasks as was the case in earlier work [7,8,35–37]. In this research I consider situations where additional security tasks (see Table 2.1 for related examples) are only allowed to have minimal or no impact on the schedule of existing RT tasks, and are not allowed to modify RT parameters. While I use specific intrusion detection mechanisms (*e.g.*, Tripwire [17], Bro [19]) to demonstrate my ideas, the proposed solutions are agnostic to the specific monitoring mechanism. The design of my scheduler-level solutions and the design of the specific security tasks are orthogonal problems. Since I aim to maximize the frequency of execution of security tasks, security mechanisms whose performance improves with frequency of execution (*e.g.*, intrusion monitoring and detection tasks or logging/tracing mechanisms) benefit from my model.

2.4 Real-Time Security Integration Mechanisms

In the following I first present security integration techniques for single core systems (Section 2.4.1) and then extend it for multicore RTS (Section 2.4.2).

2.4.1 Integrating Security: Single Core

I first propose to execute security tasks as *lower priority* than RT tasks – this is to ensure that security monitoring mechanisms can be performed without perturbing RT scheduling order. I refer to this scheme as *opportunistic execution* of security tasks. However, recall that actual periods of the security tasks are unknown and we need to *adapt* the periods within acceptable ranges to achieve better trade-off between schedulability and security.

Metric: Let T_i be the period of the security task $\tau_s \in \Gamma_S$ that needs to be determined. My goal is to minimize the perturbation between the achievable period T_s and the desired period T_s^{des}. Hence I define the metric: $\eta_s = \frac{T_s^{des}}{T_s}$ that denotes the *tightness* of the frequency of periodic monitoring for the security task τ_s and bounded by $\frac{T_s^{des}}{T_s^{max}} \leq \eta_i \leq 1$. The period selection problem is therefore to maximize the tightness of the achievable periods of all the security tasks without violating the schedulability condition (*i.e.,* WCRT is less than or equal to the deadline) and period bound constraint (*i.e.,* $T_s \in [T_s^{des}, T_s^{max}]$) of the security tasks. This monitoring frequency metric provides one way to *trade-off security with schedulability* and allows us to execute security tasks with a frequency closer to what the designers expect.

In early work [27] I develop techniques to execute security *opportunistically (i.e., with lower priority) in conjunction with RT tasks, while keeping the best possible periods using the "tighness" metric that ensures all the tasks in the system remain schedulable*. In particular, I formulate the period selection as a constraint optimization problem and solve it using geometric programming (GP) [38] approach in polynomial time.

Adaptive Security Integration: If the security tasks always execute with lowest priority, they suffer more interference (*i.e.,* preemption from high-priority RT 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 proposed a multi-mode model [28]. This framework (called Contego) allows the security policies/tasks to execute in two different *modes* (see Fig. 2.1 for a high-level schematic). For the most part, Contego executes in a PASSIVE mode with opportunistic execution of security tasks as before [27]. 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 2.4.1.1). This ACTIVE mode potentially executes with higher priority, while ensuring the schedulability of RT tasks. I also note that we need to determine periods of the security tasks for both modes as before and consider mode change overheads in the schedulability analysis.

2.4.1.1 Experience and Evaluation

Implementation: To observe the performance of the proposed scheme in a practical setup, I implemented Contego on an embedded platform (configured with 1 GHz ARM Cortex-A8 single-core processor and 512 MB RAM) [39]. I used Linux as the operating system – that allowed me to utilize the existing Linux-based security tasks for the evaluation. Since the vanilla Linux kernel is unsuitable for hard RT scheduling, I enabled the RT capabilities with the Xenomai [40] patch (kernel version 3.8.13-r72) on top of an embedded Debian Linux console image. I measured the WCET of the RT and security tasks using ARM cycle counter registers (*e.g.,* CCNT). The prototype implementation was developed in C and uses a fixed-priority

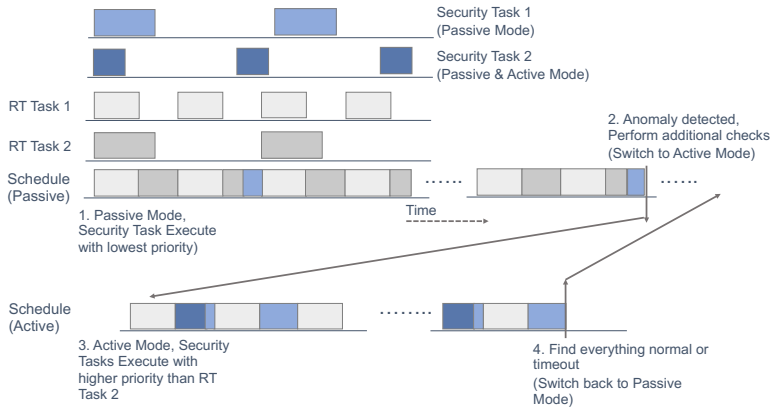


Figure 2.1: Contego: Flow of operations depicting the PASSIVE and ACTIVE modes for the security tasks.

scheduler powered by the Xenomai RT patch. The RT and security tasks were invoked by the 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 RT application, I considered a UAV control system (refer to Table 2.2). I implemented it using an open-source UAV model [41]. The original application codes were based on the STM32F4 micro-controller (ARM Cortex M4) and developed for FreeRTOS [42]. Because of differences in library support and execution semantics, I updated the source codes accordingly and ported them to Linux.

Table 2.2: RT 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 (*i.e.*, actuator commands)	5000
Reconnaissance	Read radar/camera data, collect sensitive information and send data to the base control station	10000

Security Tasks: For the security tasks, I considered two lightweight open-source intrusion detection mechanisms, *(i)* Tripwire [17], 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 [19] that monitors anomalies in network traffic. As Table 2.3 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 tasks did not exceed 50% of the total

system utilization.

Table 2.3: Security tasks used in the experiments

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

Experiment Setup: To study the detection performance I injected malicious code into the system that mimics anomalous behaviors. 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 [43] to demonstrate the attack. Malware (such as LRK, tOrn, Adore, *etc.*) in general-purpose Linux environments causes damage to the system by modifying or overwriting the system binary [44, Ch. 5]. Thus I followed a similar approach to demonstrate a host-level attack, *viz.,* I injected ARM shellcode [45] to override the victim task's code and launched the attack by modifying the contents in the file-system binary.

For each of the experiments, the work-flow 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 (*e.g.,* Bro), a mode change request was placed and the control was switched to ACTIVE mode with the corresponding ACTIVE mode tasks (see Table 2.3).

Results: I compared the performance of Contego with opportunistic execution approach that has no provision for mode changes and in which 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. 2.2. The x-axis in Fig. 2.2 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

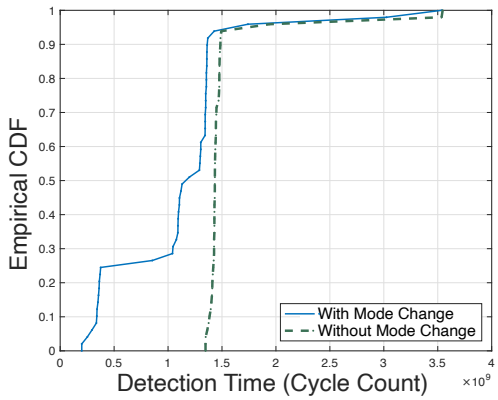


Figure 2.2: The empirical distribution of time to detect the intrusions when mode change was allowed vs when security tasks were run only in PASSIVE mode.

as $\widehat{F}_{\alpha}(\jmath) = \frac{1}{\alpha}\sum_{i=1}^{\alpha}\mathbb{I}_{[\zeta_i \leq \jmath]}$, 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 $\jmath$ represents the x-axis values (*i.e.,* the detection times in cycle count) in Fig. 2.2. The indicator function $\mathbb{I}_{[\cdot]}$ outputs 1 if the condition $[\cdot]$ is satisfied and 0 otherwise.

From Fig. 2.2 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 opportunistic execution scheme does. Recall that by opportunistic execution the security tasks are allowed to run only when other RT tasks are not running, leading to more interference (*e.g.,* 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 RT tasks, Contego had a superior detection rate in general for most of the experiments.

2.4.2 Integrating Security: Multicore

I now discuss security integrating techniques for multicore-based RT platforms. For a multicore system scheduled with partitioned scheduled scheme [32], one fundamental problem while integrating security tasks is to determine *which security tasks will be assigned to which core and executed when*. I first present a scheme with fixed core assignment for the security tasks (Section 2.4.2.1). I then extend the model where security tasks can *migrate across cores* and provide faster detection (Section 2.4.2.2).

2.4.2.1 Integration of Security Tasks with Fixed Core Assignment

Note that for multicore platforms (*i.e.*, $M > 1$) security tasks can execute in any of the M available cores. Although any period $T_s^{des} \leq T_s \leq T_s^{max}$ is acceptable, the actual *task-to-core assignments* and the *periods* of the security tasks are not known. 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 possible[1] (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 propose an iterative scheme (called HYDRA) that jointly find the core-to-task assignment and suitable periods for security tasks [29]. The key idea of HYDRA is as follows.

I start with the highest priority security task τ_s and try to obtain the best period (by solving an optimization problem) for each available core $\pi_m \in \mathcal{M}$. If there exists a set of cores $\mathcal{M}'_s \subseteq \mathcal{M}$ for which a period is obtained satisfying the RT constraints, I pick the core $\pi_{m^*} \in \mathcal{M}'_s$ that gives the maximum tightness and allocate the security task to core π_{m^*}. This will ensure that the more critical security tasks will get a period close to the desired one. I then 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 I return the taskset as *unschedulable* 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 RT tasks, if possible) in order to integrate security for the target system.

Evaluation: Recall from Section 2.2 that my goal is to explore the possible ways in which security could be integrated in multicore-based RT platforms. The HYDRA mechanism assumes that the RT tasks are distributed across *all* available cores. Another design choice available is to allocate a dedicated core for security while the RT tasks are assigned to the remaining cores. I now compare HYDRA to this alternate mechanism for security task allocation – that I refer to henceforth as the "SingleCore" allocation mechanism. Given the taskset is schedulable, one of the benefits of the SingleCore scheme is that there is no requirement for assigning security tasks. While evaluating SingleCore, all the RT tasks are partitioned into $M - 1$ cores leaving the other core for security tasks. Note that in the SingleCore scheme security tasks do not suffer any interference from RT tasks (since they use separate cores).

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

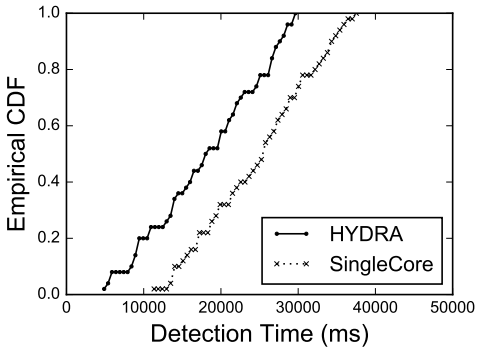


Figure 2.3: HYDRA vs. SingleCore: empirical CDF of intrusion detection time.

Case-study with a UAV Control System and Security Applications: The goal of this experiment was to observe the runtime behavior of HYDRA. I considered a quad-core system (*e.g.,* $M = 4$) running the UAV RT tasks (see Table 2.2). For security tasks I used Tripwire and Bro as before (Table 2.3). For each of the trials, I observed the schedule for 500 s and during any random time of execution we triggered attacks that corrupts the file system and network packets. I assumed that the intrusions were correctly detected by the security tasks (*i.e.,* there is no false positive/negative errors) and measured the empirical CDF of worst-case detection time. From Fig. 2.3 we can observe that paralleling security tasks across cores leads to faster intrusion detection time for HYDRA (*e.g.,* higher empirical CDF). From my experiment I found that *on average* HYDRA can provide 27.23% faster detection rate for a quad-core system. While SingleCore scheme does not experience any interference from RT tasks, however, low priority security tasks can still suffer inference from high priority security tasks. Therefore running security tasks in a single core leads to higher periods and consequently poorer detection time.

2.4.2.2 Continuous Security Monitoring

I now extend HYDRA with an alternate design mechanism that can *raise the responsiveness of such monitoring tasks by increasing their frequency of execution*. For instance, consider an intrusion detection system (IDS) *e.g.,* Tripwire – that checks the integrity of filesystems. 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. I therefore we present a design-time framework (named HYDRA-C) that enables *continuous execution* of security tasks (*i.e.,* execute as frequently as possible) across cores, without impacting schedulability of existing RT tasks [30].

One fundamental question in this scheme 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. Mathematically period selection can be expressed as: $\underset{T_s, \forall \tau_s \in \Gamma_S}{\text{minimize}} \sum_{\tau_s \in \Gamma_S} T_s$, subject to $\mathcal{R}_s \leq T_s \leq T_s^{max}, \forall \tau_s \in \Gamma_S$ where $\mathcal{R}_s$ is the WCRT of the task τ_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 therefore first derive the WCRT of the security tasks and use it as a (lower) bound to find the periods.

Period selection using HYDRA-C works as follows. I first fix the period of each security task T_s^{max} and calculate the response time $\mathcal{R}_s$. If there exists a task τ_j such that $\mathcal{R}_j > T_j^{max}$ I report the taskset as unschedulable since it is not possible to find a period for the security tasks within the designer provided bounds. If the taskset is schedulable with T_s^{max}, I then optimize the periods from higher to lower priority order. To be specific, for each task $\tau_s \in \Gamma_S$ I perform a logarithmic search [46, Ch. 6] 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, *e.g.*, $\forall \tau_j \in lp(\tau_s) : \mathcal{R}_j \leq T_j^{max}$ and repeat the search for next security task.

Experiment with an Embedded Platform: I implemented my ideas on a rover platform manufactured by Waveshare [47]. The rover peripherals (*e.g.*, wheel, motor, servo, sensor) are controlled by a Raspberry Pi 3 (RPi3) [48] single board computer. I used Linux kernel 4.9 and enabled RT capabilities by applying the PREEMPT_RT patch [49] (version 4.9.80-rt62-v7+). I performed experiments 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 RT tasks and considered two RT 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 RT task utilization was 0.7040). I introduced two security tasks: *(a)* Tripwire, that checks intrusions in the image data-store and *(b)* a custom-developed security task that checks current kernel modules (for detecting rootkits) and compares with an expected profile of modules. The WCET of the security tasks were 5342 ms and 223 ms, respectively and the maximum periods[2] of security tasks were assumed to be 10000 ms (*e.g.*, total system utilization is at least $0.7040 + 0.5565 = 1.2605$).

I compared the performance of HYDRA-C with HYDRA (Section 2.4.2.1) where I propose to statically partition the security tasks among the multiple cores. I observed the performance of HYDRA-C by

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

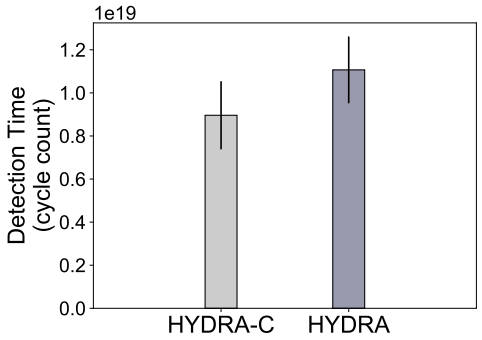


Figure 2.4: Experiments with rover platform: time (cycle counts) to detect intrusions.

analyzing how quickly an intrusion can be detected. I considered the following two realistic attacks: *(i)* an ARM shellcode [45] that allows the attacker to modify the contents of the image data-store – this attack can be detected by Tripwire; *(ii)* a rootkit [50] that intercepts all the `read()` system calls – our custom security task can detect the presence of the malicious kernel module.

In Fig. 2.4 I show the average time to detect both the intrusions (in cycle counts, collected from 35 trials) for HYDRA-C and HYDRA schemes. From this experiment I found that, on average, HYDRA-C can detect intrusions 19.05% faster compared to the HYDRA approach (Fig. 2.4). Since HYDRA-C allows security tasks to migrate across cores, it has shorter periods and that leads to faster detection times. Thus HYDRA-C subsumes the approach in HYDRA and provides faster detection.

Chapter 3

Protecting Actuators in Cyber-Physical Systems

In Chapter 2 I show that how can we integrate security tasks as a first-class principle of RT scheduling algorithms. I now present a framework to secure legacy CPS by monitoring application behaviors (*e.g.,* actuation outputs).

3.1 Overview

Since CPS are largely based on sensing and actuation, any false/spoofed command to the actuators can disrupt the normal operation of the physical plant. Commonly used open-source CPS development stacks (such as Linux) do not provide explicit control over actuation signals. For instance, if the application task obtains permission (say, root or other privileged user access) to the peripheral interface (*e.g.,* I2C [51]), it is possible to send arbitrary signals to the actuators. Let us consider an industrial robotic arm (running an embedded variant of Linux in an ARM Cortex-A53 platform [48]) that periodically opens and closes the grip to drop off and pick up objects in an assembly line. The movement of the grip is controlled by a servo. I use an open-source implementation [52] for this robotic arm where each operation is represented by a pulse value `x` (where $x = 577$ for `grip_open()` and $x = 420$ for `grip_close()`) and each pulse sends the following four 1 byte command sequences to the servo registers: `0 & 0xFF`, `0 >> 8`, `x & 0xFF`, `x >> 8`. An example of a spoofing attack for this control arm is presented in Fig. 3.1 (x-axis is the servo access sequence number and y-axis is the corresponding pulse value). Without any actuation command validation, it is possible to send arbitrary (high) pulses to the servo registers that prevents the grip from picking up/dropping objects (showing in the shaded region, see the top figure) that is not otherwise possible when my scheme (called Contego-TEE, see Section 3.4 for details) is enabled (bottom figure).

My proposed framework, Contego-TEE, prevents the sending of malicious/undesired commands to

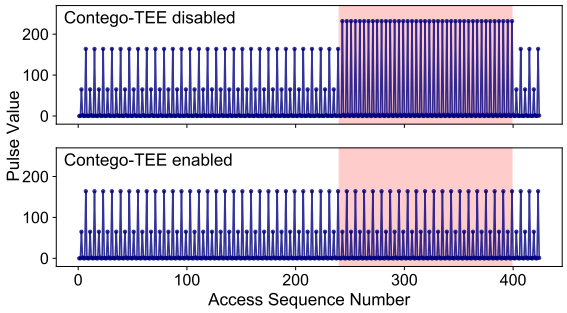


Figure 3.1: Demonstration of a control spoofing attack on a robotic control arm running embedded Linux.

physical actuators and ensures safety of the system [53]. Specifically, I use the concept of trusted execution environments (TEEs) [54] available in commodity processors (*e.g.,* ARM TrustZone [55], Intel SGX [56]) to ensure that our protection mechanisms can not be disabled even if the host OS is compromised. I develop a rule-based checking mechanism as well as design-time (schedulability) tests to ensure timing and safety requirements of the system. Contego-TEE specifically designed for *legacy systems* developed with COTS components and *does not require any modification to the application code/logic*.

3.2 Preliminaries: TEE and ARM TrustZone

TEE is a set of hardware and software-based security extensions where the processors maintain a separated subsystem in addition to the traditional OS components. TEE technology has been implemented on commercial secure hardware such as ARM TrustZone [55] and Intel SGX [56]. In this work I consider TrustZone as the building block of Contego-TEE due to wide acceptability of ARM processors for embedded systems – although my proposed framework can be ported into other TEE platforms without loss of generality.

ARM TrustZone contains two different privilege blocks: *(i) normal world* (NW) and *(ii) secure world* (SW). The NW is the untrusted environment running a commodity untrusted OS where SW is a protected computing block that only runs privileged instructions. SW in TrustZone defines the memory regions that can only be accessed by privileged instructions and the code that runs in the SW has higher privilege than the NW. Hardware logic ensures that the resources in the secure world can not be accessed from the normal world (*e.g.,* if the code running in the NW tries to access protected memory regions, TrustZone throws a hardware exception). The SW instructions are triggered when a specific flag in the processor *e.g.,* non-secure (NS) bit in the secure configuration register (SCR) is not set. These two worlds bridge via a software module referred to as *secure monitor*. The context switch between the NW and SW is performed through a *secure monitor call* (SMC).

In this work I use the TrustZone functionality to prevent the malicious commands from being sent to the actuators (refer to Section 3.4). I now present my system and adversary model.

3.3 System and Adversary Model

Let us consider a cyber-physical plant with set of RT tasks (denoted by $\Gamma = \{\tau_i, \tau_2, \cdots, \tau_N\}$). The physical system consists of a set of M actuators (*e.g.,* servo, motor, buzzer): $\{\pi_1, \pi_2, \cdots, \pi_M\}$. Each application task τ_j issues a (finite) sequence of actuation commands $\sigma_j = \{\sigma_j^1, \sigma_j^2, \cdots, \sigma_j^l, \cdots\}$ to the actuators to control physical entities (*e.g.,* wheel, propeller, alarm, robotic grip, *etc.*).

I consider the following adversarial capabilities: *(a) integrity violation* – an adversary may insert a malicious task (that respects the RT guarantees) and/or modify exiting control logic to manipulate actuator commands and control system behavior in undesirable ways; *(b) DoS* – the attacker may take control of the RT task(s) and destabilize the physical plant *e.g.,* by sending multiple control requests in a burst that may result in a malfunctioning actuator, or worse, damage the actual hardware/actuator and even threaten the safety of the system. The attacker can gain privileged (*e.g.,* root) access to perform adversarial actions (*e.g.,* to spoof control signals).

I do not make any assumptions as to how the compromised tasks enter the device. For instance, bad software engineering practices leave vulnerabilities in the systems [57]. When the system is developed using a multi-vendor model [58] (where its components are manufactured and integrated by different vendors) a malicious code logic may be injected (say by a less-trusted vendor) during deployment. The adversary may also induce end-users to download the modified source code, say by using social engineering tactics [14]. I also assume that the attackers do not have any physical access (*e.g.,* they can not physically control/turn off/damage the actuators).

3.4 Actuation Monitoring Framework

In the following I first introduce the Contego-TEE framework (Section 3.4.1). I then present mechanisms to detect any abnormal control commands issued by (rogue) tasks (Section 3.4.2) and analyze schedulability conditions that ensures my checking techniques can be enforced at runtime (Section 3.4.3).

3.4.1 Overview and Architecture

In Fig. 3.2 I illustrate the high-level overview of Contego-TEE design. Contego-TEE contains the following essential components: *(a)* a *TEE-enabled SoC* (system-on-chip) such as those supported by ARM

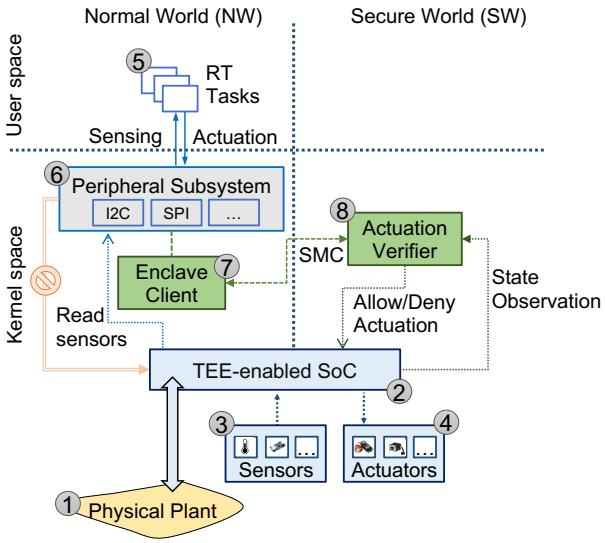


Figure 3.2: Overview of Contego-TEE system design.

TrustZone [55] (block ② in the figure); *(b)* an *enclave client* (block ⑦) that is used to communicate between NW and SW and *(c)* an *actuation verifier* (block ⑧) that is used to monitor (and validate) the actuation commands. The physical plant (①) is connected with sensors (②) and actuators (③) and controlled by the (potentially vulnerable) RT tasks (⑤). RT tasks execute in untrusted NW and issue system calls (*e.g.*, `read()`, `write()`, `ioctl()`) to access the sensors/actuators using specific interface such as I2C [51] and/or SPI [59]. Contego-TEE ensures that RT tasks cannot directly send any actuation commands (*e.g.*, it breaks the bridge between ⑥, ② and ④). I do this by placing a dispatcher (*e.g.*, enclave client) between the peripheral subsystem and actual hardware. As a result, before issuing any command to the physical actuators, it will be validated by the trusted application (*e.g.*, actuation verifier) running inside the secure enclave (*i.e.*, in the SW). In particular, when a RT task τ_i sends an actuation command x_{ik}^t to any peripheral π_k at time t, enclave client traps those request and forwards the command to the actuation verifier using SMC. Depending on the current system state $\mathcal{S}(t)$, verifier then decides whether the given command x_{ik}^t can be issued to the actuator π_k (refer to Section 3.4.2 for details). In Contego-TEE, both the enclave client and actuation verifier operate in the privileged mode (*e.g.*, kernel space) so that it can directly control low-level hardware. By using the enclave client (to invoke context switching) and verification mechanisms, Contego-TEE ensures that even if the NW RT tasks are compromised, an adversary can not send false signals to the actuators. I note that unlike NW RT tasks that may perform other computation, the actuation verifier contains a small, verified, code blocks that is used to monitor only actuation requests. I also note that Contego-TEE *does not require any application-level modifications*, *e.g.*, developers can execute unmodified, existing legacy RT tasks, using my Contego-TEE enabled kernel.

Table 3.1: Applicability of Contego-TEE for Various CPS Platforms

Platform	Application Domain	Actuators	Possible Verification Conditions*
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
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)
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
Robotic vehicle (aerial/ground)	Manufacturing, surveillance, agriculture	Servo, motor	*(a)* Check if the robot is following the mission; *(b)* allow only predefined number of actuation commands per period

*I omit mathematical expressions for readability.

3.4.2 Command Verification

For a given CPS platform, I consider the availability of an command verification function $\mathsf{CheckActuation}(\tau_i, \pi_k)$ that predicts the actuation signal and only allows access if the output of the function matches that of the requested command. In particular, if a task τ_i sends actuation command x_{ik}^t at time t to any peripheral π_k, $\mathsf{CheckActuation}(\tau_i, \pi_k)$ first obtains system state $\mathcal{S}(t)$ by observing a set of signals $S_i = \{s_1, s_2, \cdots, s_{L_i}\}$ and decides whether x_{ik}^t is valid for current state $\mathcal{S}(t)$. For example, consider a warehouse water monitoring system where an alarm is triggered only if the water level of the tank (measured by the sensor s_{WL}) is higher that a predefined threshold (θ_{WL}) and/or the water temperature (s_{WT}) is not in expected range (*i.e.,* $[\theta_{WT}^{t_1}, \theta_{WT}^{t_2}]$). I represent this as the following rule: $\mathsf{R}_W :: (s_{WL} > \theta_{WL}) \vee (s_{WT} \notin [\theta_{WT}^{t_1}, \theta_{WT}^{t_2}]) \rightarrow x = \mathsf{ON} : x = \mathsf{OFF}$, *i.e.,* Contego-TEE will only allow the sending of the high pulse (*i.e.,* $x = \mathsf{ON}$) to the alarm system (say a buzzer) only if the invariant conditions are satisfied. In Table 3.1 I summarize possible verification conditions that are applicable for various CPS platforms – however, this is by no stretch meant to an exhaustive list.

3.4.3 Timing Analysis

In order to perform checking mechanisms at runtime, we need to ensure that Contego-TEE should not cause delays and the timing requirements of RT tasks are satisfied (*e.g.,* they complete execution before deadline). I therefore develop design-time schedulability tests that ensure the taskset is schedulable [53]. For instance, the RT task τ_i is schedulable in Contego-TEE if the WCRT R_i^{TEE} is less than deadline, *i.e.,*

$R_i^{TEE} = C_i^{TEE} + I_i^{TEE} \leq D_i$, where C_i^{TEE} is the task WCET (including the time for world switching and command verification) and I_i^{TEE} is the interference from other tasks. The taskset Γ is referred to as schedulable if all the tasks are schedulable, *viz.*, $R_i^{TEE} \leq D_i, \forall \tau_i \in \Gamma$.

3.5 Evaluation

In this section I first present the implementation details of Contego-TEE (Section 3.5.1) and then show the viability of my approach using a case-study on a robotic vehicle (Section 3.5.2).

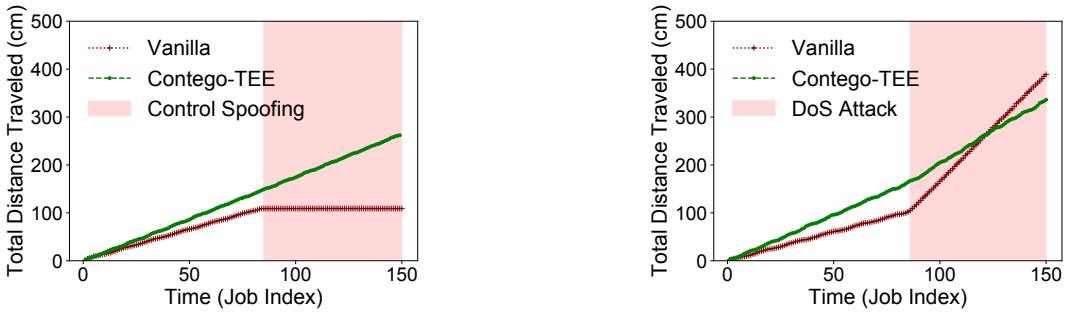
3.5.1 System Implementation

I implemented a proof-of-concept prototype of Contego-TEE on RPi3 [48] (equipped with 1.2 GHz 64-bit ARMv8 CPU and 1 GB RAM). I selected RPi3 as our implementation platform since *(a)* it supports ARM TrustZone and *(b)* previous research has shown feasibility of deploying multiple IoT-specific applications on RPi3 [14, 60–63]. I developed Contego-TEE using the open-portable trusted execution environment (OP-TEE) [64] software stack that uses GlobalPlatform TEE APIs [65] to provide TrustZone functionality. OP-TEE provides a minimal secure kernel (called OP-TEE core) that can be run in parallel with the NW OS (*e.g.*, Linux). In particular, I used Ubuntu 18.04 filesystem with a 64-bit Linux kernel (version 4.16.56) as the NW OS and the verifier is running on OP-TEE secure kernel (version 3.4). In order to implement the enclave client, I extended the Linux TEE interface (`/linux/drivers/tee/`) and enabled SMC from Linux kernel space[1]. I implemented the command verifier as an OP-TEE kernel-level trusted application (*e.g.*, in `/optee_os/core/arch/arm/pta/`). In my current implementation Contego-TEE supports actuators that are controlled via the I2C interface. Specifically, I modified the built-in structure `i2cdev_fops` (*e.g.*, in `/linux/drivers/i2c/i2c-dev.c`) with the enclave client functions that is then switch the control to the actuation verifier (*e.g.*, by using SMC).

3.5.2 Case Study: Robotic Vehicle

I implemented Contego-TEE in a COTS rover (named GoPiGo2, manufactured by Dexter Industries [66]) that can be used in multiple IoT-specific applications such as remote surveillance, agriculture, manufacturing, *etc.* [67]. The rover is equipped with two optical encoders that are connected to the motors (*e.g.*, actuator in this setup): it can turn left by switching off the right encoder and vice-versa. The detailed specifications of the rover are available on the vendor website [66].

[1]Since GlobalPlatform APIs only support SMC from user space.



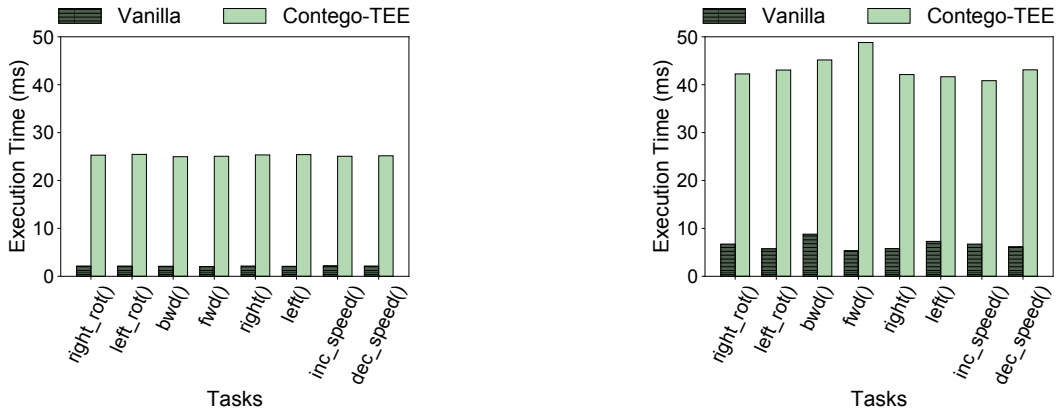
(a) (b)

Figure 3.3: Illustration of Contego-TEE under *(a)* control spoofing and *(b)* DoS attacks.

I first demonstrate how Contego-TEE can be used to protect such systems from actuation attacks and then measure the performance overheads.

Security Analysis: For the following experiments, I conducted a line following mission where the robot steered from an initial location to a target location by following a line. The controller task was running as a NW Linux application and executed vendor-provided PID (proportional–integral–derivative) closed-loop control [68] to track the planned path using the data received from sensors. The rover used the following commands: $\mathsf{fwd}()$, $\mathsf{lft}()$, $\mathsf{rht}()$, $\mathsf{st\_sp}(\delta)$ for navigating the rover forward/left/right and set the speed to δ, respectively, where each command sent a 5-byte value to the actuator registers (*e.g.,* wheel encoders/motors) using the I2C interface. For this mission, I manually inspected the vendor-provided control code and translated them into following rules that were used to monitor control signals (denoted as cmd): $\mathsf{R}_1 :: s_{LF} < -\theta \rightarrow \mathsf{cmd} = \mathsf{st\_sp}(\delta_1) \wedge \mathsf{rht}()$, $\mathsf{R}_2 :: s_{LF} > \theta \rightarrow \mathsf{cmd} = \mathsf{st\_sp}(\delta_1) \wedge \mathsf{lft}()$ and $\mathsf{R}_3 :: s_{LF} \in [-\theta, \theta] \rightarrow \mathsf{cmd} = \mathsf{st\_sp}(\delta_2) \wedge \mathsf{fwd}()$ where s_{LF} was the readings from the sensor, $\theta = 2500$ was a vendor-provided threshold (*e.g.,* to follow the line) and $\delta_1, \delta_2 \in [0, 255]$ were used to set the speed of the rover.

The x-axis of Fig. 3.3a shows the time (*e.g.,* count of the controller job) and the y-axis is the total distance travelled by the rover (*e.g.,* readings from the encoders). In order to demonstrate malicious behavior, I followed a strategy similar to that considered in prior work [12,67,69,70]. In particular, during program execution, I injected a logic bomb (during the shaded region in Fig. 3.3a) and sent erroneous commands to the controller. In this case, during the control spoofing attack, the rover deviated from the mission (*e.g.,* PID control loop) and falsely sent commands to turn off one of the motors. As a result, when Contego-TEE was not active, the rover was not following the line and the encoder readings (*i.e.,* traversed distance) remained same (see the maroon dashed line in the figure). I next executed the same code with Contego-TEE enabled (green curve in the figure). In this case, when each control command was issued,



(a) (b)

Figure 3.4: Runtime of rover control tasks with and without Contego-TEE: *(a)* for 99-th percentile and *(b)* worst-case.

my checker followed the checking conditions (*e.g.*, $R_i, 1 \leq i \leq 3$) and sent desired commands to the motors (and hence the rover was moving as expected).

In the following experiment (Fig. 3.3b), when the DoS logic bomb was triggered (shaded region in the figure) the task sent multiple requests to increase the speed of the rover. When Contego-TEE was not enabled, this caused the rover to move faster and hence there was a rapid increase in the encoder readings (*e.g.*, maroon line, shaded region in the figure). In contrast, when Contego-TEE was active (green line), it disallowed multiple increase speed requests per period and hence the rover followed the line with a steady speed.

Overhead Analysis: To measure the runtime overheads I conducted experiments with the vendor-provided control tasks [66] as a benchmark (Fig. 3.4). In this setup the control verifier ignored more that one actuation request per period (200 ms). The x-axis of Fig. 3.4 shows the control tasks and y-axis represents execution time *(a)* when Contego-TEE is not enabled (dark bar) and *(b)* with Contego-TEE enabled (light bar). I present the timing results for 99th percentile (Fig. 3.4a) and worst-case (Fig. 3.4b). The timing values were measured using the Linux `clock_gettime()` system call with `CLOCK_MONOTONIC` clock parameter and we present data from $10,000$ trials. As we see from the figure, Contego-TEE increases the execution time – this is expected due to (world) context switching as well for invariant checking. From the experiments I found that Contego-TEE increases execution times by *(i)* 34.11 to 43.47 ms (worst-case), *(ii)* 22.87 to 23.31 ms (99-th percentile) and *(iii)* 19.55 to 19.60 ms (average-case) for the various control tasks and hence can be used with 15 Hz (or slower) controllers (for this setup). This extra overhead results in increased security and I expect this could be acceptable for various CPS platforms.

3.6 Proposed Work

Development of Command Verification Algorithm: While Contego-TEE provides the designers the ability to verify actuation commands, the engineers need to specify rules that can be checked at runtime. Therefore, any condition/rule that is not configured during system design will not be detectable in the present framework. To address this issue, I propose to develop an *automated verification process* that can provide a systemic way to verify actuation commands for the target application. In prior work, researchers propose sequence based approaches (*e.g.,* clustering and pattern monitoring techniques) to verify different signals such as system calls [14] and network packets [71]. In theory, Contego-TEE can leverage such techniques to verify actuation commands. However, given the constraints of embedded CPS, any computationally-heavy approach *(i)* may jeopardize timing/safety constrains and *(ii)* requires additional implementation/porting efforts (due to limited library support available in the TEEs); and therefore may not be suitable for legacy systems. In addition, sequence based approaches can only verify program "execution context" and may not be suitable for scenarios where an adversary blocks actuation commands (*e.g.,* introduce intentional delay between consecutive commands). For example, consider a valid command sequence with the following timestamps (where t_i represents timestamp, σ_i denotes the command invoked at t_i and $t_{i+1} > t_i$): $(t_1, \sigma_1), (t_2, \sigma_2), (t_3, \sigma_3), (t_4, \sigma_4), (t_5, \sigma_5), \cdots$. Now, if an attacker blocks the command σ_4 and issues it at a later time (say at t_6), this will results in the following sequence: $(t_1, \sigma_1), (t_2, \sigma_2), (t_3, \sigma_3), (t_6, \sigma_4), \cdots$. Notice that for both cases the program outputs same command sequences, *i.e.,* $\sigma_1, \sigma_2, \sigma_3, \sigma_4, \cdots$ and hence sequence-based techniques such as those proposed in literature (see the related survey [72]) will not be effective to detect this attack. In the proposed research, I intend to develop techniques that can analyze *both execution and temporal contexts* to verify actuation commands. Recall that, Contego-TEE designed for periodic RT tasks. In future work I will relax this assumption and extend Contego-TEE for *general-purpose CPS* where application tasks could be event-triggered (*i.e.,* not periodic).

Implementation and Demonstration: I implemented Contego-TEE in Linux/OP-TEE and demonstrate the feasibility of this work in a RPi3-based ground rover. In future work, I intend to extend my implementation/evaluation (both the current one and future extensions) for multiple COTS-based CPS platforms – this is to demonstrate the viability of my proposed approach in various CPS applications. In addition to the rover platform described in Section 3.5.2, I will demonstrate my proposed techniques in the following platforms.

- ***Surveillance System*** *[application: home/industrial monitoring]*

  In this setup, the operator monitors a remote location – the application rotates the camera using servos (*e.g.,* actuators) and captures images/videos for remote monitoring. I will develop this CPS plant using RPi3 board and pan-tilt camera mounting system.

- ***Robotic Arm*** *[application: manufacturing]*

  This platform will demonstrate the behavior of an assembly line (see Section 3.1). For this experiment I intend to use an off-the-shelf 6-degree-of-freedom robotic arm [73].

- ***Infusion/Syringe Pump*** *[application: health-care]*

  Using this test-bed I intend to perform a medical application case-study where the CPS platform (*e.g.,* infusion pump) is used to inject fluids/drugs. I will use RPi3 board and linear actuators (mobilized by DC/stepper motors) to prototype this setup.

3.6.1 Research Task Summary

The estimated time to complete this research is six to eight months. This research includes the following work items:

- *Analytical analysis and design-space exploration:* I will study related work and design the actuation verification algorithm – this includes *(a)* development of the analytical model and *(b)* study the performance with synthetic workload/data-set to observe the effectiveness of the proposed algorithm.

- *Development of test-beds:* In parallel with analytical analysis, I will also *(a)* assemble the test-beds for experiments, *(b)* install necessary library and software packages, *(c)* analyze existing source codes and program behaviors for better understanding of the platforms and *(d)* ensure their functional correctness.

- *Implementation:* Once the model and test-beds will be available, I will *(a)* implement the proposed method (in Linux/OP-TEE running on RPi3), *(b)* port my implementation into the CPS platforms and *(c)* study runtime behavior (*e.g.,* detection accuracy and performance overhead).

- *Documentation:* I will finally document my findings and write a draft paper that targets security conferences.

Chapter 4

Related Work

Enhancing security in time-critical CPS is an active research area. In recent years researchers proposed various mechanisms to provide security guarantees into legacy and non-legacy RTS (both single and multicore platforms) in several directions, *viz.,* integration of security mechanisms [27–29], authenticating/encrypting communication channels [7–10,37,74], side-channel defence techniques [35,58, 75–77] as well as hardware/software-based frameworks [11–15,70,78]. The above mentioned work requires *(a)* modification to the scheduler or RT task parameters or *(b)* additional porting efforts/architectural support. Besides majority of the solutions are designed for single core platforms only. Therefore, it is not straightforward to retrofit those approaches for multicore legacy systems.

Perhaps the closest line of work to Contego-TEE is PROTC [62] where a monitor 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 Contego-TEE, PROTC is limited for specific applications (*e.g.,* aerial robotic vehicles) and requires a centralized control center to validate/enforce security policies. The hardware/software-based mechanisms and architectural frameworks [11, 12, 14, 15, 70] are not designed to protect against control-specific attacks and may not be suitable for systems developed with COTS components. There also exist large number of research that use TrustZone to secure traditional embedded/mobile applications (too many to enumerate here, refer to the related surveys [55,79]) – however the consideration of time-critical and control-centric aspects of RT CPS applications distinguish Contego-TEE from other research.

Chapter 5

Proposed Research: Summary and Timeline

The research tasks that I proposed in this proposal are summarized below. Figure 5.1 shows the estimated timeline for the proposed work items.

Task	Year 2020							
	May	June	July	August	September	October	November	December
W1								
W2								
W3								
W4								
W5								
W6								

Figure 5.1: Timeline of the proposed research tasks.

W1. Develop the verification algorithm and test the ideas with synthetic workload.

W2. Assemble the test-beds.

W3. Study existing source codes/program behavior and ensure functional correctness of the build systems.

W4. Implement the detection techniques (including the verification algorithm from W1) in RPi3 running Linux and OP-TEE.

W5. Customize the implementation for different platforms and port into each of the target systems. Perform experiments to study security and overhead.

W6. Document the findings and prepare a draft of the paper.

References

[1] M. Abrams and J. Weiss, "Malicious control system cyber security attack case study–Maroochy water services, Australia," *McLean, VA: The MITRE Corporation*, 2008.

[2] 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.

[3] 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*, pp. 447–462, 2010.

[4] S. S. Clark and K. Fu, "Recent results in computer security for medical devices," in *MobiHealth*, pp. 111–118, 2011.

[5] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, p. 6, 2011.

[6] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the ukrainian power grid," *SANS Industrial Control Systems*, 2016.

[7] 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.

[8] T. Xie and X. Qin, "Improving security for periodic tasks in embedded systems through scheduling," *ACM TECS*, vol. 6, no. 3, p. 20, 2007.

[9] V. Lesi, I. Jovanov, and M. Pajic, "Network scheduling for secure cyber-physical systems," in *IEEE RTSS*, pp. 45–55, 2017.

[10] V. Lesi, I. Jovanov, and M. Pajic, "Security-aware scheduling of embedded control tasks," *ACM TECS*, vol. 16, pp. 188:1–188:21, 2017.

[11] 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 international conference on High confidence networked systems*, pp. 65–74, ACM, 2013.

[12] 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*, pp. 21–32, 2013.

[13] 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*, pp. 1–6, 2015.

[14] 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*, pp. 191–196, 2017.

[15] 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*, pp. 10–21, 2018.

[16] J. Song, G. Fry, C. Wu, and G. Parmer, "CAML: Machine learning-based predictable system-level anomaly detection," in *IEEE CERTS*, pp. 12–18, 2016.

[17] "Tripwire." `https://github.com/Tripwire/tripwire-open-source`.

[18] "AIDE." `http://aide.sourceforge.net/`.

[19] "The Bro network security monitor." `https://www.bro.org`.

[20] M. Roesch, "Snort - lightweight intrusion detection for networks," in *USENIX Conf. on Sys. Admin.*, pp. 229–238, 1999.

[21] L. L. Woo, M. Zwolinski, and B. Halak, "Early detection of system-level anomalous behaviour using hardware performance counters," in *DATE*, pp. 485–490, 2018.

[22] V. M. Weaver, "Linux perf_event features and overhead," in *IEEE FastPath*, 2013.

[23] "OProfile." `http://oprofile.sourceforge.net/`.

[24] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM CSUR*, vol. 41, no. 3, p. 15, 2009.

[25] 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.

[26] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, 2011.

[27] 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*, pp. 123–134, 2016.

[28] 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*, pp. 23:1–23:22, 2017.

[29] 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*, pp. 225–230, 2018.

[30] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, "Period adaptation for continuous security monitoring in multicore systems," in *DATE*, 2020.

[31] 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.

[32] J. Chen, "Partitioned multiprocessor fixed-priority scheduling of sporadic real-time tasks," in *Euromicro ECRTS*, pp. 251–261, 2016.

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

[34] X. Wang, N. Hovakimyan, and L. Sha, "L1Simplex: Fault-tolerant control of cyber-physical systems," in *2013 ACM/IEEE ICCPS*, pp. 41–50, 2013.

[35] 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.

[36] 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.

[37] K. Jiang, P. Eles, and Z. Peng, "Optimization of secure embedded systems with dynamic task sets," in *DATE*, pp. 1765–1770, 2013.

[38] 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.

[39] "BeagleBone Black." https://beagleboard.org/black.

[40] "Xenomai – real-time framework for Linux." https://xenomai.org.

[41] "UAV control codes." https://github.com/Khan-drone/flight-control.

[42] "FreeRTOS." http://www.freertos.org.

[43] "FTP brute-force attack trace." https://github.com/bro/bro/blob/master/testing/btest/Traces/ftp/bruteforce.pcap.

[44] *Ethical Hacking and Countermeasures: Secure Network Operating Systems and Infrastructures*. EC-Council, 2nd ed., 2017.

[45] "Linux ARM shellcode." https://www.exploit-db.com/exploits/21253/.

[46] D. E. Knuth, *The art of computer programming: sorting and searching*, vol. 3. 1997.

[47] "Alphabot2 wiki." https://www.waveshare.com/wiki/AlphaBot2-Pi.

[48] "Raspberry Pi." https://tinyurl.com/rpi3modelb.

[49] L. Fu and R. Schwebel, "Real-time Linux wiki." https://rt.wiki.kernel.org/index.php/rt_preempt_howto. [Online].

[50] "Linux rootkit." https://github.com/crudbug/simple-rootkit.

[51] "I^2C manual," 2003.

[52] "Robot arm control." https://github.com/tutRPi/6DOF-Robot-Arm.

[53] M. Hasan and S. Mohan, "Protecting actuators in safety critical IoT systems from control spoofing attacks," in *ACM IoT S&P*, pp. 8–14, 2019.

[54] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *IEEE Trustcom/BigDataSE/ISPA*, pp. 57–64, 2015.

[55] S. Pinto and N. Santos, "Demystifying ARM TrustZone: A comprehensive survey," *ACM CSUR*, vol. 51, no. 6, p. 130, 2019.

[56] V. Costan and S. Devadas, "Intel SGX Explained," *IACR Crypt. ePrint Arch.*, no. 086, pp. 1–118, 2016.

[57] 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*, pp. 1–6, 2017.

[58] 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*, pp. 271–282, 2015.

[59] "SPI block guide V04.01," 2004.

[60] L. Cheng, K. Tian, and D. D. Yao, "Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks," in *ACM ACSAC*, pp. 315–326, 2017.

[61] R. Liu and M. Srivastava, "VirtSense: Virtualize Sensing through ARM TrustZone on Internet-of-Things," in *ACM SysTEX*, pp. 2–7, 2018.

[62] R. Liu and M. Srivastava, "PROTC: PROTeCting drone's peripherals through ARM trustzone," in *ACM DroNet*, pp. 1–6, 2017.

[63] T. Liu, A. Hojjati, A. Bates, and K. Nahrstedt, "Alidrone: Enabling trustworthy proof-of-alibi for commercial drone compliance," in *IEEE ICDCS*, pp. 841–852, 2018.

[64] "Open Portable Trusted Execution Environment." https://www.op-tee.org/.

[65] "TEE client API specification v1.0." https://globalplatform.org/specs-library/tee-client-api-specification/.

[66] "GoPiGo." https://github.com/DexterInd/GoPiGo.

[67] 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*, pp. 574–585, 2018.

[68] "Dexter Industries Sensors." https://github.com/DexterInd/DI_Sensors.

[69] 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*, pp. 801–816, 2018.

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

[71] M.-K. Yoon and G. F. Ciocarlie, "Communication pattern monitoring: Improving the utility of anomaly detection for industrial control systems," in *USENIX SENT*, 2014.

[72] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection for discrete sequences: A survey," *IEEE TKDE*, vol. 24, no. 5, pp. 823–839, 2010.

[73] "6-DoF robot arm." https://files.banggood.com/2016/09/6-DOF-Instructions.pdf.

[74] T. Xie, A. Sung, and X. Qin, "Dynamic task scheduling with security awareness in real-time systems," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pp. 8–pp, IEEE, 2005.

[75] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. B. Bobba, "Real-time systems security through scheduler constraints," in *Euromicro ECRTS*, pp. 129–140, 2014.

[76] 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*, pp. 1–12, 2016.

[77] K. Krüger, M. Völp, and G. Fohler, "Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems," in *EUROMICRO ECRTS*, vol. 106, pp. 22:1–22:17, 2018.

[78] D. Lo, M. Ismail, T. Chen, and G. E. Suh, "Slack-aware opportunistic monitoring for real-time systems," in *IEEE RTAS*, pp. 203–214, 2014.

[79] W. Li, H. Chen, and H. Chen, "Research on ARM TrustZone," *ACM GetMobile*, vol. 22, no. 3, pp. 17–22, 2019.