Defending Real-Time Systems through Timing-Aware Designs

Tanmaya Mishra

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Thidapat Chantem, Chair
Ryan Gerdes
Ning Zhang
Eli Tilevich
Guoqiang Yu

April 22, 2022
Arlington, Virginia

Keywords: Real-time systems, Security, Trusted Execution, CAN, CFI, Mixed Criticality
Copyright 2022, Tanmaya Mishra

Defending Real-Time Systems through Timing-Aware Designs

Tanmaya Mishra

(ABSTRACT)

Real-time computing systems are those that are designed to achieve computing goals by certain deadlines. Real-time computing systems are present in everything from cars to airplanes, pacemakers to industrial-control systems, and other pieces of critical infrastructure. With the increasing interconnectivity of these systems, system security issues and the constant threat of manipulation by malicious external attackers that have plagued general computing systems, now threaten the integrity and safety of real-time systems. This dissertation discusses three different defense techniques that focuses on the role that real-time scheduling theory can play to reduce runtime cost, and guarantee correctness when applying these defense strategies to real-time systems. The first work introduces a novel timing aware defense strategy for the CAN bus that utilizes TrustZone on state-of-the-art ARMv8-M microcontrollers. The second reduces the runtime cost of control-flow integrity (CFI), a popular system security defense technique, by correctly modeling when a real-time system performs I/O, and exploiting the model to schedule CFI procedures efficiently. Finally, the third studies and provides a lightweight mitigation strategy for a recently discovered vulnerability within mixed criticality real-time systems.

Defending Real-Time Systems through Timing-Aware Designs

Tanmaya Mishra

(GENERAL AUDIENCE ABSTRACT)

Real-time computing systems are those that are designed to achieve computing goals within certain timing constraints. Real-time computing systems are present in everything from cars to airplanes, pacemakers to industrial-control systems, and other pieces of critical infrastructure. With the increasing interconnectivity of these systems, system security issues and the constant threat of manipulation by malicious external attackers that have plagued general computing systems, now threaten the integrity and safety of real-time systems. This dissertation discusses three different defense techniques that focuses on the role that real-time scheduling theory can play to reduce runtime cost, and guarantee correctness when applying these defense strategies to real-time systems.

The first work introduces a novel timing aware defense strategy for the Controller Area Network (CAN). CAN is a popular communication system that is at the heart of every modern passenger vehicle and is indispensable for the safe operation of various components such as the engine and transmission systems, and due to its simplicity, may be vulnerable to a variety of attacks. We leverage security advancements in modern processor design to provide a lightweight and predictable (in terms of time taken to perform the operation) defense technique for some of these vulnerabilities.

The second work applies a technique called Control-Flow Integrity (CFI) to real-time systems. CFI is a general-purpose defense technique to prevent attackers from modifying software execution, and applying such techniques to real-time systems, particularly those with limited hardware capabilities, may be infeasible. By applying real-time scheduling theory,

we propose a strategy to apply CFI to such systems, while reducing its overhead, or cost, without compromising the security guarantees CFI inherently provides.

Finally, safety-critical systems may consist of a mix of operations, each having a different level of importance (criticality) with respect to the safe operation of the system. However, due to the complexity of modeling such systems, the models themselves may be vulnerable to attacks. Through simulations we study one such vulnerability and propose a modification to mitigate it.

Dedication

To my grandfather who taught me how to work hard and have fun

Acknowledgments

This work was made possible through the concerted effort and guidance from many people, over the course of many years. I would first and foremost like to thank my advisor, Dr. Thidapat Chantem. Without her guidance, criticism, encouragement and invaluable advice, over more than half a decade, all of this work would have been impossible. I will forever be indebted to her for the countless lessons, technical and otherwise, that I have learned from her.

I would also like to thank my co-advisor, Dr. Ryan Gerdes, for his (many) criticisms and guidance during my Ph.D. journey, Dr. Ning Zhang for his technical comments and advice, and Dr. Guoqiang Yu and Dr. Eli Tilevich for their advice as part of my advisory committee.

There have been many others who have helped me in this journey, maybe even too many to list here. However, some of the most influential people, in both my growth as a person and this work, have been my lab mates and colleagues. Foremost of them has been Pratham, a person to bounce ideas off, and constant friend (and roommate) over the last five and a half years. Many of the ideas presented in this work are from late night conversations in the lab or over a session of chai, with others including Gaurang and Mahsa, at the Minerva Express Indian grocery store. The daily cups of chai at this store will always remain some of my brightest memories from my Ph.D. journey. Over the years, many have left and even more have joined these sessions, including Spandan, Rajarshi and Pragya. I am thankful for their support and glad that this tradition will pass on to them as they go through their respective academic journeys.

While those at the lab have been constant sources of support, it would be a remiss on my part if I do not thank my friends from my undergraduate days. These include Gaurang, Aniket,

Prathamesh, Anurag, Niharika, Raj, Karan, Omkar, Amogh, Madhura, among others. The numerous trips we have had together, over the years, have been the bright moments of respite that were sorely needed to maintain the motivation and spirit required to complete this journey. For over a decade, these people have always remained a constant in my life and I will always be grateful for them. Special thanks to Gaurang and Raj for the time spent in Seattle during the summer of 2021. That time was a great breath of fresh air. I am grateful to Madhura for her constant support, encouragement and for being a source of happiness through out the years.

Finally, I would like to thank my parents, brother, and my cousins, Ankita and Ambika, for their constant support and motivation through the years, and Anushila (and, of course, Priyanka) for being my home away from home that I could always escape to when needed.

Contents

List of Figures xiii

List of Tables xv

1 Introduction 1

1.1 Motivation . . . 2

1.2 Limitations of embedded system hardware . . . 6

1.3 Predictability of real-time systems and its usefulness for system security . . . 7

1.4 Reviewing advancements in real-time system theory for security vulnerabilities 8

1.5 Organization and contributions . . . 9

2 Preliminaries: TEE and ARM TrustZone for Cortex-M 13

2.1 ARM TrustZone for Cortex-M . . . 14

3 Utilizing trusted execution to secure CAN bus communications 18

3.1 Introduction . . . 18

3.2 Related Work . . . 22

3.3 Preliminaries . . . 24

3.3.1 Controller Area Network (CAN) . . . 24

3.4 System Model and Problem Statement 26
3.4.1 Real-Time Task Model 26
3.4.2 Threat Model 27
3.4.3 Problem Statement 28
3.5 System Design and Overview 29
3.6 TEECheck : A TEE based CAN message checker 31
3.6.1 Transmission 32
3.6.2 Reception 38
3.7 Experimentation 39
3.7.1 Experimental Setup 40
3.7.2 Results 41
3.8 Analyses 45
3.8.1 Real-Time Analysis 45
3.8.2 Security Analysis 46
3.9 Conclusion 48

4 Utilizing hard real-time system predictability to implement control-flow integrity 49

4.1 Introduction 49
4.2 Related Work 53
4.3 Preliminaries 55

4.3.1 Hardware model 55
4.3.2 Software Model 56
4.3.3 Threat model 57
4.4 Overview of Procrastinating CFI 58
4.5 Procastinating CFI Task model 59
4.5.1 Application task model 61
4.5.2 Security task model 63
4.6 Security Task Deadline Relaxation 65
4.7 Ensuring Correctness and Schedulability 68
4.7.1 Race condition between output and security tasks 68
4.7.2 Implicit data-dependency between output and security jobs 70
4.7.3 Implications on scheduling 70
4.8 Procrastinating CFI mechanism 72
4.8.1 SAU based function-block enforcement and shadow stack 73
4.8.2 RTOS modifications 74
4.8.3 Design Alternatives 75
4.8.4 Verification of Control Flow in Security Task 76
4.9 Security analysis 76
4.10 Evaluation 78
4.10.1 Control flows in Cyber Physical System (CPS) Software 78

4.10.2 Experimental setup 79

4.10.3 Hardware Overhead 79

4.10.4 Simulation study 81

4.11 Conclusion 83

5 Simulation-based characterization and a priority degradation-based defense against the Mad Monk attack for mixed-criticality systems 84

5.1 Introduction 84

5.2 Related Work 87

5.3 Mixed Criticality Task Model 90

5.4 The Mad Monk Attack 92

5.4.1 Threat Model 93

5.4.2 Mad Monk Attack mechanism 94

5.5 Simulating Mad Monk 96

5.5.1 Simulator setup 96

5.5.2 Determining basic conditions with realistic penalty 99

5.5.3 Changing to harmonic periods 100

5.5.4 Considering variations in penalty on a newly released intermediate victim job 101

5.5.5 Converting the attacker task into a sporadic task 103

5.5.6 A note on dropping attacker jobs for single and multiple attacker tasks - suicide condition 107

5.5.7 Simulation summary 109

5.6 Criticality level-aware priority degradation to mitigate Mad Monk 110

5.6.1 Response time analysis for degraded group for RM 115

5.6.2 Simulating effect on deadline misses for attacker tasks 117

5.7 Conclusion 117

6 Conclusion and Future Work **118**

6.1 Summary and Conclusion 118

6.2 Future Work - Vulnerability in real-time systems due to self-suspension . . . 119

Bibliography **121**

List of Figures

2.1 ARMv8-M microcontroller power ON code flow 14

3.1 CAN 2.0b standard frame format 26

3.2 (a) Regular ECU system and (b) ECU system utilized for TEECheck 30

3.3 (a) Source verification using HMAC (b) Rate limiting messages based on per-task last transmission time 33

3.4 TEECheck Reception scheme 38

4.1 (i) Task system where sensor task has simple in-line CFI, (ii) Proposed procrastinating model where in-line CFI is bundled into a security task (Section 4.5.2) which can implement complex CFI operations (accomodate greater WCET) but with pushed back deadlines to lower resource utilization. 58

4.2 Code redirection on a generic microcontroller. Attacker modifies contents of `R0` to change branch target. 60

4.3 Overview of Procrastinating CFI mechanism to capture forward-edge control-flow logs (reading source and destination addresses of branch) and backward-edge verification. Branching into upgraded memory launches fault handler that performs both operations - Section 4.8. 62

4.4 Race condition: Attacker is able to execute and affect input of output task. In such a case, the output task's deadline (and output release time under LET) is before the security task is able to complete execution. 67

4.5 (i) Sample data dependency between internal tasks A and C with output tasks B and D. (ii) Dependencies when security tasks are considered. 69

4.6 SAU based function-call enforcement. 72

5.1 Victim here is the intermediary victim job. Periodic jobs are harmonic synchronous. Sporadic attacker job ”skips” a previous job, leading to an advantageous condition. 107

5.2 Criticality based degradation strategy implemented on a 3 criticality level system. Each criticality change reduces the number of possible attackers that can target a HI criticality intermediary victim task. 110

List of Tables

3.1 Single task running at highest frequency (RL - Rate Limiting, CAN - CAN controller transmission time) 40

3.2 Single task running at highest frequency with reception. TEECheck call overhead 42

3.3 Transmitting different message sizes 44

3.4 Automotive benchmark with increasing number of tasks with TEECheck . . 46

4.1 Control-flow transfers in popular real-time applications. 79

4.2 Deadline relaxation of security tasks. 80

4.3 Percentage of tasksets (/10,000) that are only schedulable under EDF+SRP when push backs are accounted. Original task set utilization does not include security tasks. WCET Ratio is ratio of WCET of security tasks to application task. 82

5.1 Simulation where average execution time is 60% of worst-case for given criticality level. Success % is the number of task sets (out of 100) that show successful attack. 98

5.2 Simulation where average execution time is 80% of worst-case for given criticality level. Success % is the number of task sets (out of 100) that show successful attack. 100

5.3 Changing to harmonic periods immediately increases the attack success ratio by 3 times. Success % is the number of task sets (out of 100) that show successful attack. 101

5.4 Success percentages when attacker cannot induce penalty on newly released intermediary victim jobs. Success % is the number of task sets (out of 100) that show successful attack. 102

5.5 Attacker can induce up to 10% of the penalty on the newly released intermediary victim job, that it can otherwise induce on a preempted job. Success % is the number of task sets (out of 100) that show successful attack. 103

5.6 Attacker can induce up to 70% of the penalty on the newly released intermediary victim job, that it can otherwise induce on a preempted job. Success % is the number of task sets (out of 100) that show successful attack. 104

5.7 Comparing success rate of periodic and sporadic attacker for different number of tasks in task set. Success % is the number of task sets (out of 100) that show successful attack. Earliest Successful Attack % is where a task set shows strictly earlier successful attack occurrence under the specific type of attacker. 105

5.8 Success % where only the sporadic task was able to achieve successful attack 106

5.9 Simulating criticality aware priority degradation on LO criticality attacker all other tasks MID or HI criticality. Deadline Miss % is the number of task sets (out of 100) where attacker task misses its deadline. 116

Chapter 1

Introduction

Real-time computing systems are those that must maintain logical correctness while providing temporal guarantees. Specifically, these systems define a relation between when the system is provided an input and by when it must genenerate an output. The effect or consequence of violating this definition varies. For *soft* real-time systems the consequence of violating this pre-defined relation, also called *deadlines*, may simply degrade the quality of service (QoS) for the system user. An example of such a system is when a user streams a video from a remote server (such as YouTube or Netflix). If the user computer is unable to handle decrypting and rendering the video stream on screen in a timely manner, or if the network is overloaded, the user may have to wait time for the video to render and/or download and see frequent stutters while viewing the video stream. On the other hand, in the case of *hard* real-time systems, violating deadlines during system runtime could lead to catastrophic system failure and the consequences could include loss of human life. For example, the airbag in modern passenger vehicles must deploy within strict time bounds or else it would be too late to prevent injury to the occupants in the vehicle, rendering the airbag deployment system ineffective. Therefore, such systems have *hard deadlines* which must not be violated. While traditionally such computing systems have been isolated and work autonomously, due to the rise of system interconnectivity, such systems are becoming increasingly vulnerable to malicious external influence. This dissertation specifically looks at the increasingly relevant problem of security vulnerabilities in modern embedded

real-time system design and aims to motivate the larger real-time systems research community to consider security objectives and vulnerabilities alongside traditional real-time system guarantees. This dissertation shows that even in computationally limited systems, there is a possibility to include defenses against external attacks by judiciously utilizing real-time system concepts and techniques. Further, this disseration showcases that, due to the predictability that is inherent in real-time systems, modern real-time system models need to be rethought and updated to mitigate vulnerabilities that they may inadvertently introduce, due to their design.

1.1 Motivation

This work is based on two pillars: real-time system scheduling theory and mechanisms, and system security. An interesting observation of *all* real-time systems is the *inherent predictability* of these systems which cannot be said of an arbitrary computing system. Since real-time systems have deadlines which are known apriori to deployment, it is but evident that such systems will perform useful work, that can be accurately measured, within these deadlines. Therefore, such a system's state can be accurately judged at any given time instant. This leads to an interesting relationship between temporal guarantees and security when such systems are judged by existing research and literature in the system security domain. However, before we discuss this relationship, the reader may question as to why is it necessary to consider the security of real-time systems in the first place? Why is this a relevant problem in the current state-of-the-art?

Consider, for instance, a vehicle from around 40 years ago. These were relatively simple machines and would have an internal combustion engine, a simple manual transmission, hydraulic brakes, and maybe a microprocessor-based fuel injection system [78] for improving

engine fuel efficiency and engine output. Therefore, the computing systems within such vehicles were simple and robust to ensure occupant safety, and, more importantly, completely isolated from other computing systems. In fact, at that time, the internet was still being used primarily by scientific and military organizations for specialized work. With the proliferation of communication technologies over the last 3 decades and its application to computing environments ranging from massive data-centers to the personal smartphone, coupled with an ever increasing data bandwidth and transfer speeds, it is but natural that some of these technologies would trickle into critical systems such as vehicles. Modern vehicles are now far more complex machines. While the mechanics of the internal combustion engine may not have made major progress over the years, highly sophisticated add-on systems such as hybrid/electric drivetrains and advanced driver assistance systems (ADAS) have become increasingly popular in modern vehicles. ADAS mechanisms such as traction control, adaptive cruise control, automatic pedestrian-detection etc., utilize data captured by a vast array of sensors, computed in over a 100 distributed computing systems, also called electronic control units (ECU) spread throughout the vehicle that constantly communicate over intra-vehicular networks such as controller area network(CAN) [49]. Since the safety of the occupants as well as that of pedestrians on the road is dependent on the correct and timely operations of such systems, such ADAS systems have *hard real-time* requirements [25] to ensure occupant safety.

More recently, vehicles have been slowly opened up to external connections too. For example, in the last decade, it has become increasingly common to have wireless connectivity options such as Bluetooth, WiFi and cellular connectivity integrated into the vehicle's infotainment system for passenger convenience as well as for remote over-the-air (OTA) updates to vehicle software. In fact, manufacturers such as Tesla are capable of fine-tuning core vehicle performance such as top speed and efficiency via such OTA updates. While this has obvious

advantages for those that are able to purchase such vehicles, the ability to perform such modifications to core vehicular functionality have already been successfully exploited. For example, researchers [85] were successfully able to send out malicious messages on a Tesla's CAN bus via a malicious OTA payload to perform potentially dangerous operations, such as disabling the vehicle's power steering and braking. While Tesla later sent out an OTA software patch after being notified, it can be argued that such attacks are only going to get more sophisticated with the passage of time.

Unfortunately, security vulnerabilities plague real-time systems in other domains too. For example, Iran's nuclear reactor program was severely undermined by the Stuxnet [48] computer worm which was able to modify plant operations. In fact, Stuxnet showcases an even more disturbing problem. Stuxnet was able to spread and maliciously take control even though the plant control computers were *air-gapped*. That is, they were physically isolated from external computing systems just like traditional real-time systems. However, the human operators were able to load the worm into the controllers by plugging in infected storage drives. Infected Stuxnet controllers could have caused catastrophic plant failure, and resulted in possibly Chernobyl-like consequences to human and animal life. Further, since these systems run legacy software which are rarely, if ever, updated since they control critical equipment, the possibility of such software having unpatched vulnerabilities is higher. As the sophistication of such attacks increase, legacy and connected modern real-time systems alike will become more vulnerable.

Therefore, there is an outstanding need to build security mechanisms into real-time systems. While taking reactionary measures, such as that performed by Tesla, to safeguard such systems is a possibility, it entirely depends on the benevolence of the attacker to inform real-time system vendors about such security vulnerabilities. However, Stuxnet shows that attackers exist who would try to exploit such systems for a myriad of reasons, from to

economic to political, regardless of the consequences of such actions, and would only be detected by either forensic analysis of such systems or when the attack is executed in its entirety. The former, however inconvenient, is still more desirable to the latter which could have disastrous consequences.

To summarize this discussion, and determine the various facets this dissertation must consider in the domain of securing real-time systems, we can circle back to the example of a modern connected vehicle. The salient points are:

1. ECUs within modern connected vehicles have limited computational resources to manage size, weight and power consumption (SWaP) while also balancing the bill of materials (BoM) of the final product. Designing a system security solution must explicitly consider hardware constraints, especially in an embedded system scenario.

2. ECUs (and many other real-time systems that are deployed in critical environments) are increasingly vulnerable to advancements in system security attacks. Introducing general-purpose system security defenses in such hardware constrained systems may lead to insurmountable overheads, which may lead to system designers completely abandoning the use of any such security solution. Existing general-purpose designs must be judiciously modified to fit the specific needs of real-time systems to convince system designers to introduce these security solutions in real-time systems. In fact, it could be advantageous to utilize some of the assumptions of real-time systems to reduce the cost of existing security solutions.

3. Advancements in real-time systems research may inadvertently introduce more vulnerabilities. It is imperative, due to the increasing interconnectivity of modern real-time systems such as in the context of modern vehicles, to critically view recently introduced real-time systems models and determine if they allow malicious external manipulation,

and fix these problems as they arise.

We shall now discuss each of these salient points, and then conclude the introductory material with the specific contributions of this dissertation.

1.2 Limitations of embedded system hardware

While it is clear from current state of cyber-physical system security that there is a need to harden real-time systems to reduce the possibility of malicious manipulations, the type of hardware that is used in such systems must be explicitly considered.

Unlike desktop or server-grade computing environments that one usually encounters in domains such as cloud computing, high-performance computing, etc., many real-time systems utilize less complex hardware. For example, the ECUs used within vehicles typically use CPUs that share more similarities with microcontrollers with integrated RAM, Flash and typically run at much lower clock speeds of a few 100 MHz [72] as compared to state-of-the-art CPUs which easily execute at least a magnitude higher clock speeds. This is done to consider size-weight-and-power (SWaP) constraints, especially for systems that are deployed in remote or inaccessible locations where dependability and power consumption are more important requirements than raw performance. In fact,the recently launched Perseverance rover for Mars exploration uses a radiation-hardened IBM RAD750 processor [21] which itself is 2 decades old and based on a processor design from the late 1990's. The performance of such systems that run critical real-time software is a far cry from what is available in modern processors. Further, many microcontrollers have much simpler architecture and lack modern features such as complex memory management units. Such systems also use simple operating systems called real-time operating systems (RTOS) such as FreeRTOS [11] that

are built to be highly predictable and provide a small hardware-abstraction layer (HAL) to simplify application development while introducing the least possible additional software components.

Therefore, where appropriate, this dissertation strictly considers embedded systems which utilize low-performance microcontrollers for processing environments. Any technique presented, that specifically requires certain hardware capabilities, assumes that the technique will be deployed on modern variants of common low-performance microcontrollers that are currently available off-the-shelf. Such hardware introduce their own set of limitations which must be carefully considered while designing the real-time system software. It is assumed that the software running on such systems is either written to be bare-metal or, more commonly, divided into individual *tasks* that are scheduled via the RTOS.

1.3 Predictability of real-time systems and its usefulness for system security

As stated before, real-time systems are highly predictable. While this is necessary for correct and safe system operation, it makes it that much easier for attackers to exploit such systems. For example, real-time systems allow for timing inference attacks [36] where task timing characteristics are used to leak information regarding task behavior during runtime. In fact, a novel attack method has been developed for a modern real-time task model that depends on expected system behavior due to the model's design, and, within this thesis, a countermeasure is proposed to mitigate this vulnerability.

On the other hand, the very same predictability can be used to inform defense mechanisms to make them more performant and/or reduce their cost on the entire system. Since system

behavior is predictable, for example, it is known apriori the number of messages sent out on the CAN bus by an ECU task in a modern vehicle, it is possible to inform defense mechanisms to consider typical system behavior to both a) harden the system against attacks, and b) lighten the load on the underlying hardware. In fact, it is even possible to perform increased security countermeasures on the same hardware while maintaining deadlines by exploiting the predictability of the system. However, care must be taken when exploiting predictability to reduce the cost of security countermeasures. Any assumption of predictability must be built upon the careful consideration of how the system would operate if the predictability assumption does not hold true due to malicious interference or if, within the scope of the attack, if the assumptions cannot be violated by the attacker.

Note that, in the theme of this dissertation, any designs and ideas for improving the integration of general-purpose system security defenses in real-time systems must consider the limitations of the underlying embedded hardware. Judiciously exploiting any architectural advantages afforded by modern variants of such types of hardware, alongside integrating traditional real-time system concepts, may help to make the final design tenable for typical embedded real-time systems.

1.4 Reviewing advancements in real-time system theory for security vulnerabilities

While the previous two sections deal with how modern embedded hardware constrains the ability of system designers to integrate system security solutions, and how existing real-time theory and techniques may counterbalance these constraints, it would be remiss to not discuss whether the advancements in real-time system theory introduce new system vulnerabilities.

Traditionally, real-time systems research involves modelling such systems to improve resource utilization and efficiency, while strictly maintaining safety and predictability. However, due to the increasing interconnectivity of modern real-time systems, it is not feasible anymore to build these systems without considering malicious interference. There is currently a nascent trend in this domain of research that specifically looks at vulnerabilities within state-of-the-art real-time task models. These vulnerabilities do not lie within system *implementations* but instead lie within the *modelling* of expected system behavior. It could be argued that such flaws are more dangerous than any single implementation flaw, since any correctly implemented systems, that closely match such flawed models, are automatically vulnerable.

Therefore, it is imperative to look at prior work that aims to break the assumptions underlying real-time system models, and/or utilizes the assumptions in these models to force the system to behave in an unintended manner. Such vulnerabilities must then be carefully addressed by fixing the assumptions within the model. This dissertation looks at one such vulnerability that has been discovered by prior work, and proposes a revised model to address this issue.

1.5 Organization and contributions

An overview of the rest of this dissertation is now presented. As discussed above, it is clear that there exists an avenue to apply the predictability of real-time systems to general system security concepts and that real-time system modeling may be flawed and open to attacker manipulation. Additionally, where applicable, the proposed defenses must recognize the constraints of the hardware on which it must operate.

Therefore, this dissertation is organized in a progressive manner. It starts by introducing a technique that is designed to co-depend on the underlying hardware architecture, and

timing constraints of the system, to aid the defense mechanism. The second work reduces the cost (and improve the capabilities) of a well-known system security defense mechanism by carefully utilizing real-time modeling techniques. Here, certain underlying hardware capabilities are assumed only to provide an example implementation of a defense mechanism. Finally, the dissertation discusses a modification to a set of recently introduced real-time task models to harden them against a vulnerability discovered, by prior work, within these models. This is done by completely eschewing the requirement for specific hardware requirements, and by integrating the defense into the model's design itself.

The techniques are summarized here for the reader:

- **Chapter 3** - This chapter focuses on utilizing modern embedded architecture to design a defense mechanism against a set of general security vulnerabilities, while being mindful of the system's timing requirements. Specifically, the CAN bus in vehicles is considered. Modern embedded hardware architectures, allow strong isolation between execution contexts through the introduction of architecture extensions such as ARM's TrustZone [12]. Such advancements are now available in the lowest powered micocontrollers and are built to introduce very low overheads and very *predictable* into system operation. The work presented in this chapter considers the temporal tightness of a typical hard real-time workload which would be found in a vehicle ECU. The work presented aims to showcase that it is possible to design defenses, even on low-end hardware, that defends against masquerade and denial-of-service attacks as well as prevents information leakage. The chapter presents experimental proof on real hardware that shows that such a technique can benefit ECU software without requiring large modifications to the original code base and does not introduce enough overhead to violate the temporal guarantees of pre-existing code.

- **Chapter 4** - While the previous chapter essentially presents a system design perspective to solve the hard real-time system security vulnerabilities utilizing predictability in hardware operation. This chapter, on the other hand, delves into a specific attack, collectively called *control-flow attacks* that are used to manipulate a computing system to perform attacker-controlled arbitrary computations, with a minimal set of requirements. While control-flow attacks and their defense mechanisms have been studied for over a decade, prior literature either does not validate the applicability of presented techniques to hard real-time systems, or does not consider the unique limitations of resource-constrained hardware commonly found in hard real-time systems. This chapter presents a novel approach to that utilizes real-time theory and, especially, assumptions regarding when a system receives input and by when it generates output. Such assumptions are made possible due to the *predictability* of the operation of the system, making it possible to temporally *spread* load by delaying control-flow transfer checks. This allows for not only improved utilization of system resources (that are already scarce in systems utilizing resource-constrained embedded hardware), but also opens an opportunity to integrate more complex defenses that are currently only available for systems with more capable hardware. Care is taken to ensure that the design does not violate the *correctness* of the defense mechanism and does not inadvertently introduce additional blindspots that could be exploited by an attacker.

- **Chapter 5** - This chapter delves into studying and defending against the recently proposed Mad Monk attack for mixed criticality systems. Mixed criticality task models have been proposed over the last decade to improve resource utilization in safety critical systems. Mixed criticality task models allow the system designer to explicitly configure the hierarchy of tasks by their *importance* to the safe and correct execution of the system. Importance, or *criticality*, of a task is a concept that may be orthogonal to

the traditional concept of *priority* in real-time systems since the priority of tasks may be determined with respect to other temporal parameters (such as deadline or frequency) depending on the scheduling algorithm, and may have no relation to how critical the task is to system operation. In mixed criticality systems, system criticality changes if a high criticality task requires greater execution time (suggesting non-average behavior), allowing the scheduler to assign greater execution budgets to higher criticality tasks while degrading lower criticality tasks. This re-shuffling of execution budgets allows the system to not only efficiently utilize system resources when system operates in the average condition, but also quickly re-prioritize system resources when the system enters non-average operating conditions. Mad Monk exploits this expected system behavior and can cause degradation of higher criticality tasks via a lower criticality task. In this chapter, by studying the conditions under which this attack can take place through extensive simulations, a new defense is proposed by modifying the degradation strategy.

Chapter 2

Preliminaries: TEE and ARM TrustZone for Cortex-M

Trusted Execution Environments (TEEs) are trusted execution context within an untrusted execution environment. TEEs are widely used for Digitial Rights Management (DRM) of copyrighted audio and video content in modern smartphones. For example, Google's Widevine DRM [107] is a popular framework that depends on the existence of a TEE to provide highest Quality-of-Service (QoS). For instance, streaming services could refuse to provide the highest quality of audio/video data if they do not detect a TEE or detect a compromised TEE. To ensure the integrity of TEEs and their constituent software, also called Trusted Applications (TA), there needs to mechanism to separate and enforce untrusted and trusted software components. TEEs are built to address a powerful threat model, i.e., an attacker that can take control over not only the application software but more privileged components such as the operating system itself. Therefore, TEEs must be supported by hardware extensions which constrain the attacker to the untrusted memory and execution space, and safely switch between untrusted and trusted spaces with manageable overheads. One such hardware mechanism is ARM's TrustZone.

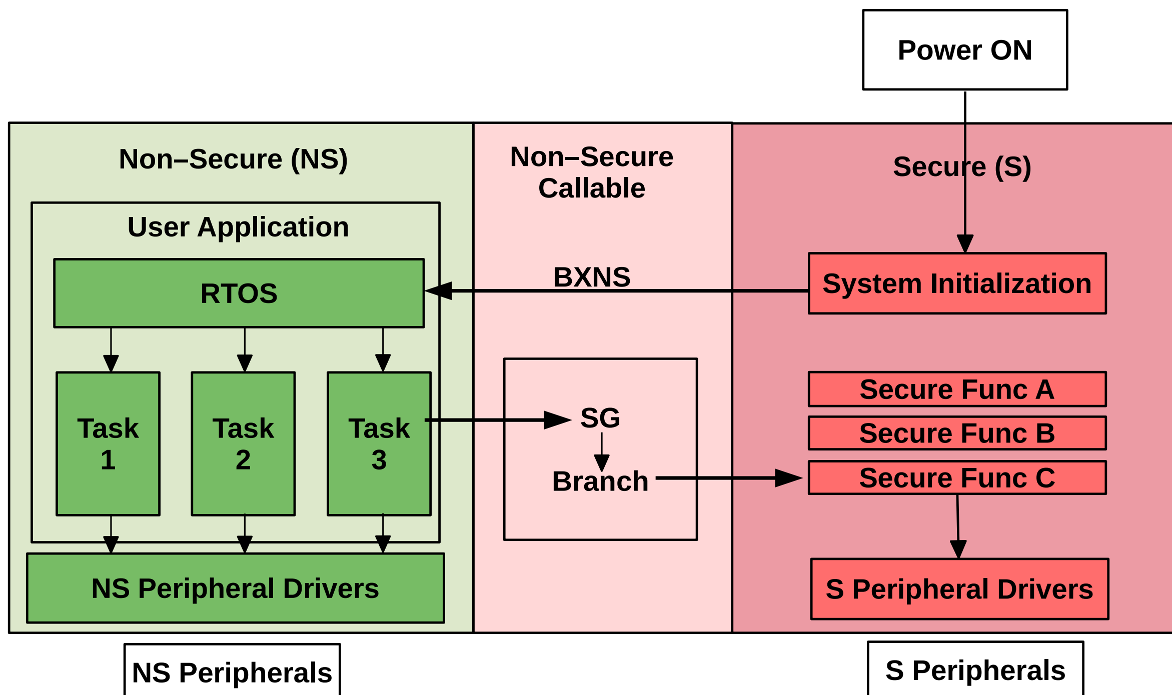


Figure 2.1: ARMv8-M microcontroller power ON code flow

2.1 ARM TrustZone for Cortex-M

ARM TrustZone for Cortex-M [109] (based on ARMv8-M architecture) is a variant of the TrustZone technology first introduced in ARM's Cortex-A processors. ARM TrustZone is a set of processor architecture extensions which allow creating TEEs via software. It divides the processor execution into two domains, **secure** and **non-secure**. Code running in the secure domain has access to information from both domains while code running in the non-secure domain has access to information only from the non-secure domain. While TrustZone for Cortex-A is complex and has significant overheads [84], TrustZone for Cortex-M is designed to be very lightweight. To reduce the overhead for a low-powered microcontroller to switch between the two states, TrustZone for ARMv8-M utilizes a near static memory-mapped mechanism for delineating the domains. The TrustZone divides the memory space such that certain addresses are made available only to the secure domain. This is facilitated through

a hard-wired controller logic called the implementation defined attribution unit (IDAU). The IDAU creates a striated memory partitioning scheme such that it is easy to identify to which domain an address belongs. Specifically, if the 29th bit of the memory address is 0, it is a secure domain address. Additionally, certain sections of the non-secure domain can be upgraded to the secure domain through software using the security attribution unit (SAU). Depending on the implementation, peripherals are memory-mapped into both secure and/or non-secure memory locations. The non-secure peripheral locations are enabled via a peripheral access controller (PAC) or security control unit (SCU). The SAU, PAC, and SCU are themselves mapped to secure locations by the IDAU, making it impossible for non-secure code to access or modify them, unless TrustZone is broken.

The secure code memory location is further divided into secure (S) and non-secure Callable (NSC) locations. While the former cannot be accessed by any code running in the non-secure domain (or it would cause the system to generate a hard fault likely requiring human intervention), the NSC locations provide an intermediary jump point where the secure gateway (`SG`) instruction is kept which switches the processor mode to secure when executed. All calls into the secure side have the interface function defined in the NSC. From an execution point of view, both domains have a *Thread* and *Handler* mode for regular and interrupt code executions, respectively. If an interrupt is generated from the non-secure side and the secure code is currently executing, all information is pushed to the secure stack and registers are cleared before the switch to the non-secure interrupt handler. The same set of steps happen when the situation is reversed. Based on our experiments (Section 3.7), the fact that both domains have the same execution flow and capabilities in ARMv8-M allows for consistently low interrupt latency regardless of the domain from which the interrupt originates (4 us overhead for switching in our case).

Figure 2.1 shows the flow of code execution once an ARMv8-M controller is powered on.

Code execution for ARMv8-M processors begins in the secure domain, which then branches into the non-secure domain. The bulk of the application code is written to run in the non-secure domain, including a real-time operating system (RTOS), task code and peripheral drivers in our case[1]. When required, the application task code makes calls to the secure code via the intermediary functions present in the NSC.

It should be noted that the secure and non-secure domains are orthogonal to the regular processor privilege levels. Within each domain, the processor still executes under the traditional privilege model, where interrupts and the RTOS may run in privileged processor execution mode while task code may run under unprivileged processor execution mode. Further, there may be a shared or separate memory protection unit (MPU) for the secure and non-secure domains. An MPU is accessible from the privileged execution mode and enforces fine-grained access rights to certain memory locations for privileged and unprivileged code. Privileged code can access memory locations not specified in the MPU table, while access from unprivileged code would generate a fault. It must be noted the SAU has a very similar operation to that of the traditional microcontroller MPU but they are separate entities that can work together. Specifically, the SAU is used to augment the partitioning of the memory space into secure and non-secure domains over and above the fixed partitioning scheme provided by the IDAU while the MPU works within this partitioned memory space to provide different access right to different pieces of code. For example, an RTOS can load task-specific access rights before context switching to a task. This is possible because the SAU supports multiple *regions* depending on the implementation. Each region is a set of registers where secure domain code can programmatically upgrade a non-secure domain address space to secure domain. Switching between the domains has low overhead which is shown by the results in

[1] While it is possible to run all the code inside the secure domain (within space limitations), it is notoriously difficult to produce bug-free code on a larger scale [96]. An attacker with knowledge of vulnerabilities in the secure code could compromise the entire system since the secure code has access to the entire memory space.

Chapter 3. Non-secure domain code cannot read/write contents of secure domain memory and can call only specially marked *non-secure callable* (NSC) secure domain code. Any other cross-domain memory access from the non-secure domain causes a high priority hard-fault. It should be noted that, traditionally, the hard-fault on ARM processors was utilized to handle critical system exceptions. As part of the ARMv8-M *mainline* architecture (implemented in the Cortex-M33 architecture), an illegal cross-domain memory access (TrustZone violation) is handled by a dedicated `SecureFault` exception handler. However, we aim to target even the ARMv8-M *baseline* architecture (implemented in the lower-powered Cortex-M23 variant) that bundles a TrustZone violation into a hard-fault, and calls the `HardFault` exception handler, to save on manufacturing costs. Throughout the rest of this thesis, we refer to either of the fault exceptions as a hard-fault exception for simplicity.

Our solution in Chapter 3 uses a combination of the MPU and careful partitioning of resources using the TrustZone. We utilize the SAU and the hard-fault for the procrastinating CFI defense mechanism in Chapter 4.

Chapter 3

Utilizing trusted execution to secure CAN bus communications

3.1 Introduction

Today's vehicles are complex machines. While they still have the same basic design of the internal combustion engine and transmission as vehicles from decades ago, they now have sophisticated, highly automated features such as advanced fuel injection systems, hybrid drivetrains, traction control, adaptive cruise control, and automatic lane-keeping, all of which are supported by sensor data and processing units. In addition, vehicles have become even more like regular computing systems, with (1) remote software updates that improve performance, or (2) having a certain degree of autonomy, allowing it to drive itself for short distances. To support these functionalities, vehicles must now generate, process and act upon a large amount of information to make driving safer, more comfortable, and more efficient.

Vehicular control systems are distributed throughout the vehicle, with some located physically close to the sensors and actuators with which they interact. While there have been proposals to consolidate the various components into a centralized system [10, 104] that controls every aspect of the vehicle, most vehicles still utilize separate electronic control units (ECUs) that are dedicated to specific functions. Modern vehicles have upwards of 100 different ECUs and this number is constantly increasing as vehicle manufacturers add func-

tionality. While one ECU may control the engine, others may control the vehicle entertainment system, dashboard information, brakes, fuel system, etc. Advanced driver assistance mechanisms act upon information, in real-time, from many of these ECUs simultaneously. For example, certain luxury modern vehicles have crosswind stabilization which adjusts the vehicle braking characteristics under strong crosswinds. To do so, information from sensors measuring wind speed, steering position and characteristics (steering column ECU), and vehicle speed (engine control unit), among others, is processed in real-time to provide safe braking assistance.

To achieve timely sharing of vehicle runtime information between ECUs while reducing the size, weight, and power constraints (SWaP) and manufacturing cost, modern cars utilize a shared bus system for inter-ECU communications. In this chapter, we consider the CAN bus, an industry standard protocol for intra-vehicular networks. There is a large body of work in both academia and industry, along with the millions of cars that utilize it, which show that CAN is an efficient and robust communication network. While the older CAN protocol may not be sufficient for the autonomous vehicles of the near future, a newer variant (CAN-FD [60]) has been designed to increase the longevity of the protocol. However, with the increasing amount of data being shared over the bus, it becomes critical to develop techniques to not only maintain bandwidth availability but also the real-time nature of message transmission.

In addition to efficiency and timeliness, security has become an important consideration. As vehicles increasingly make decisions autonomously to ensure passenger comfort and safety, it becomes imperative that their operations are not disrupted. Due to the critical role of the CAN bus in facilitating and maintaining safe and reliable vehicle operation, CAN may draw heavy interest from malicious actors who wish to take control of a vehicle or change operational characteristics to make it unsafe for its riders. In fact, prior work [35, 69] has

already shown various attacks that can take place on a vehicle, allowing malicious parties to perform actions such as applying brakes and causing a crash. For instance, newer vehicles have additional communication interfaces such as WiFi, Bluetooth and cellular connectivity to connect to external servers for software updates or for passenger convenience. These are usually provided by ECUs that are often directly connected to the vehicle's CAN bus. Such external interfaces may be vulnerable to exploitation and become a gateway to access the vehicle. For example, it has been shown that the Bluetooth stack in the car infotainment unit has vulnerabilities that when exploited, allow attackers to run arbitrary code on an ECU [35]. The attacker could then use this compromised ECU to perform masquerade attacks [19] where the attacker poses as a legitimate entity and sends out spoofed messages that could affect and/or control critical parts of the vehicle, e.g., sending messages to cut off fuel supply to the engine, or launch denial-of-service (DoS) attacks by flooding the bus with garbage messages. This is possible since CAN is a broadcast bus without message authentication. If such attacks are carried out in real-world scenarios, such as in high-speed traffic, it could have catastrophic consequences.

To maintain predictable and timely operation on the CAN bus, we believe that the best form of defense would be at the source itself. That is, an attacker that *fails to utilize* the bus is less effective than one which is given access to the bus and may have the ability to disrupt message transmissions. this chapter builds upon this core idea and proposes a lightweight technique to secure the CAN bus from attacks such that the compromised ECU cannot engage the bus more than the original (uncompromised) ECU.

To do so, we propose that ECUs utilize trusted execution environment (TEE)-capable processors which allow code compartmentalization, making it possible to verify CAN message source and destination within an ECU itself instead of at the receiver ECU. TEEs are isolated execution environments designed to run trusted software and are supported by processor ar-

chitecture extensions which include strict access control policies to processor components, peripherals, data, and address buses. TEE implementations such as Intel SGX [40] and ARM TrustZone [12] can currently be found in commercially available hardware and have been used in a variety of security-critical applications, such as Samsung Pay [13]. In addition, TEE is a known entity, as there exists a large body of work [92] which studies the security benefits and pitfalls of TEEs. Although our approach requires changes to the critical hardware components inside a vehicle, we believe it to be an especially effective one, both in terms of performance and security. In fact, our approach incurs *no* increase in CAN bus bandwidth consumption and we observe substantial performance improvements over currently available approaches while running on much slower (12 MHz clock instead of 100+ MHz clock for typical ECUs) hardware. The shift towards using CAN-FD over CAN in recent vehicles [1] shows that the automotive industry is willing to utilize newer technologies when significant advantages are demonstrated. This chapter has the following major contributions:

1. We propose a TEE-based ECU system architecture to separate message generation and consumption from CAN transmission and reception.

2. Based on our new system architecture, we present **TEECheck**, an intermediary CAN bus interface which achieves efficient and trustworthy vetting of message origin and frequency before message transmission, and of request origin before received message data is disbursed. In particular, TEECheck:

    (a) Detects and prevents an attacker from *masquerading* as another legitimate message source such as other tasks on the same *or* different ECU.

    (b) Provides a *proactive* and *on-ECU* mechanism to mitigate DoS attacks on the CAN bus. Our technique is the first to utilize TEE to contain the aforementioned attacks to the compromised ECU and does *not* require a receiver ECU to validate

whether a message is from a legitimate party.

   (c) Provides an *on-device* mechanism to prevent an attacker from *any* access to messages not intended for it, i.e., prevents snooping. To the best of our knowledge, there are no other techniques designed for the CAN bus that prevents the attacker from accessing the CAN message frames meant for other endpoints.

3. We experimentally show that the overhead associated with our approach, which is incurred *only* when a task requires access to the CAN bus, is fairly negligible and quite predictable, making our approach suitable for resource-constrained devices running real-time applications. Specifically, the overhead typically takes 477 us from message generation to message transmission, and 480 us for message data reception, on a 12 MHz processor which would translate to 50 µs on a 100+ MHz processor.

While we consider CAN 2.0 as our target application in this chapter, our mechanism can be easily adapted to any broadcast communication mechanism, such as I2C or SPI, with minimal changes. In terms of TEE implementation, we selected the ARM TrustZone for Cortex-M (ARMv8-M architecture with Security Extensions) [109] which, although fairly recently introduced, has a number of commercial-off-the-shelf microcontroller implementations available that can be used in modern ECUs today as drop-in replacements.

We shall now look at relevant prior work.

3.2 Related Work

CAN bus security has become an important research area in the past several years. Considering the safety-critical nature of the systems where CAN is utilized, e.g., automobiles, this is not surprising. CAN bus hardening approaches are spread across the communication

stack layers. At the physical layer, intrusion detection systems (IDS), have been introduced. These schemes are variants of clock-skew and voltage-based fingerprinting which help to detect and, in some cases, identify attacker ECUs. While such techniques can detect an attacker within a few frames [38, 39, 108] or single frame [50], they still, in general, require a specialized monitoring node for detection and are *reactive* in nature. That is, they only detect when the attacker has managed to engage the bus.

Network-level authentication techniques for CAN bus have also been widely studied. LeiA [91] and vatiCAN [87] are AUTOSAR [51] compliant authentication schemes which utilize some form of MAC based authentication at the receiver end. To compensate for the increased overhead, others such as CANAuth [102] and LiBrA-CAN [54] are based on variants of CAN+ [114], a backward-compatible variant of CAN capable of higher data rates, or CAN-FD-Sec [34] based on the CAN-FD [60]. Unlike CAN+, CAN-FD is expected to be the next-generation replacement for CAN 2.0. All these works require receiver-end authentication, which incurs unnecessary bus overhead since illegitimate messages must still be transmitted before they can be authenticated.

In addition, these work cannot prevent DoS attacks since receiver-end authentication cannot stop a transmitter from sending out a message. In contrast, we limit the frequency of message transmission of the compromised ECU. Further, we aim to be CAN variant agnostic. Our other goal is to propose a lightweight hardware based approach which requires minimal changes to the ECU software. In contrast, Berg et al. [20] considers separating the infotainment system from the rest of the vehicle by implementing secure gateways, which requires significant software addition and can incur large time overhead.

The concept of trust has been used to fortify CAN bus communication in prior work. VeCure [106] uses the concept of trust groups where ECUs handling critical operations are kept in the higher trust group, authenticating each other using a MAC based scheme. The work

by Gui et al. work [58] is more similar to ours in that it utilizes hardware trusted platform module (TPM) for establishing a root of trust. Perhaps the closest work, in spirit, to ours is VulCAN [100] where the authors utilize trusted execution. Their technique builds upon LeiA and vatiCAN by utilizing a trusted computing base (TCB, in their case, Sancus [86]) inside which they generate their MACs. VulCAN takes much longer, i.e, around 2ms for the entire authentication sequence. In addition, addressing DoS attacks are outside the scope of all of these techniques since they depend on the receiver for attack detection.

We shall now discuss some preliminary information regarding CAN bus before we begin discussion regarding the System Model.

3.3 Preliminaries

We now provide an overview of the underlying technologies of our system. Specifically, we consider the ARM TrustZone for ARMv8-M (Section 2.1) based microcontrollers and the CAN protocol. Due to the page limit, we only provide the details that are relevant to our work. For more details, readers are referred to existing publications [49, 109].

3.3.1 Controller Area Network (CAN)

CAN is a protocol originally designed for communication between different vehicle components but has been applied to other areas such as industrial automation. It is a serial communication protocol designed to broadcast small messages over a shared bus. CAN uses a multi-master communication paradigm where nodes compete, on a per-message basis, to send messages on the bus and it is upto each node to accept or ignore the messages.

Currently, vehicles utilize the CAN 2.0 version which supports bitrates up to 1 Mbits/sec,

although there has been ongoing effort to introduce the newer CAN-FD (CAN flexible data rate) [60] variant which allows larger message data (64 bytes instead of 8 bytes) and higher bitrates of up to 5 Mbits/sec. The CAN 2.0b standard frame format is shown in Figure 3.1. CAN supports different types of frame formats, including a DATA frame which contains the actual payload for communication between nodes, a REMOTE frame which is sent from a node requesting data from another node, an ERROR frame which is sent from a node when it detects an error on the bus, and an OVERLOAD frame to provide an additional delay between successive DATA or REMOTE frames. While we concern ourselves, in this work, with the DATA frame format since it contains the actual data being transmitted on the bus, our solution can be extended to consider REMOTE frames with minimal modifications.

Bus contention is resolved using the arbitration field. DATA frames may have different arbitration field lengths–11 bits for the *Standard Frame* (shown in Figure 3.1), and 29 bits for the *Extended Frame* formats (extended frame format is only valid for DATA or REMOTE frames). These arbitration bits constitute the message IDENTIFIER and is used by receiver nodes to identify if the message pertains to them. All nodes which wish to transmit sense the bus for any ongoing transmission and back off when they detect one. When the nodes sense that the bus is free to use, they send a start of frame (SOF). All contentions are resolved as the nodes send the arbitration bits. If a node senses a dominant bit (logic 0) being transmitted as it is transmitting a recessive bit (logic 1), it loses arbitration and stops transmission. Receiver CAN controllers utilize message identifier masks for filtering messages sent out on the bus. The bitmasks are applied to message identifiers as they are made available on the bus. If there is no match, the CAN controller stops listening on the bus until it detects a SOF on the bus. These bitmasks are programmed and kept in the CAN controller memory for comparison when a message IDENTIFIER is received.

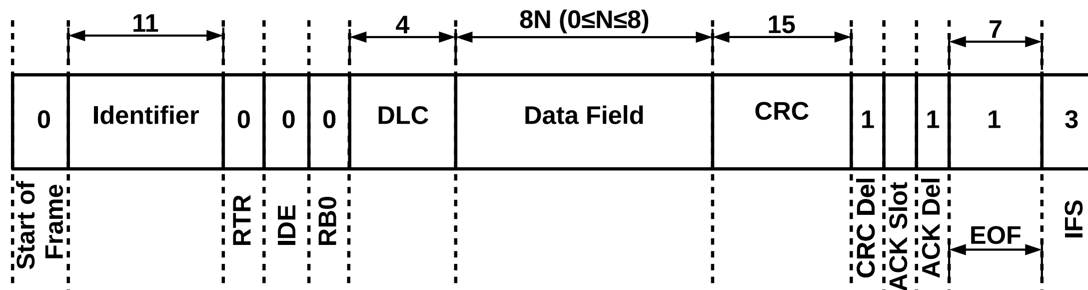


Figure 3.1: CAN 2.0b standard frame format

3.4 System Model and Problem Statement

We now discuss the real-time task model, threat model, and formally define the problem in this section.

3.4.1 Real-Time Task Model

We model all ECU tasks as periodic real-time tasks, each of which is described by a worst-case execution time (WCET) and a period. The period of these tasks can be lower-bounded based upon the time it takes to generate a message for transmission. Without a loss of generality, we assume that each task in the ECU is responsible for generating CAN messages with *non-overlapping* identifiers. That is, each task has a set of identifiers associated with it and only it. However, this is not a limitation of our approach and multiple tasks can share the same identifiers if so required. Further, certain ECUs may contain *emergency* tasks. For example, for airbag deployment, the airbag control module, arguably one of the most time-sensitive ECUs in a vehicle, may generate an emergency hard sporadic task to send a message to cut-off fuel to the engine to prevent a fire in the event of a crash. We consider such a task as having a very high priority, and which would be able to preempt any currently running periodic ECU task. We assume that tasks are scheduled using a priority-based round-robin scheduling algorithm where a higher-priority task always preempts a lower-

priority task and processor time is equally divided between tasks of equal priority. Such a policy is selected for ease of implementation and predictability. In fact, ARM's commercial RTOS, Keil RTX5 [77], which we utilize in our experiments, uses this policy.

3.4.2 Threat Model

We consider a threat model where the attacker has *remote* access to a vehicle ECU. For detecting physical intrusions, such as an attacker attaching a malicious ECU to the bus, authentication must be done at the network level or at the receiver, which has already has been addressed by prior work [34, 54, 87, 91, 106] and is beyond the scope of this work. We consider that the attacker has taken advantage of external network interfaces made available by certain ECUs on the vehicle (such as WiFi and Bluetooth network interfaces created by a vehicle's infotainment unit) and has managed to compromise task(s) on the ECU. We also assume that the attacker operates *only* within the non-secure domain, and an attacker-controlled task can still make calls to the API made available in the NSC region. We assume two attacker privilege cases:

1. Base case - The attacker takes control of task(s) (for example, through return-oriented programming [95]) on an ECU, is able to execute arbitrary code under this context, and can generate messages with identifiers meant for other tasks on the same, or different, ECU. The attacker, however, is unable to escalate its privilege level to match that under which the RTOS is executing. We believe that an attacker can be restricted to this privilege level since our system utilizes strict memory access guards using a hardware MPU when running any unprivileged non-secure code which limits its ability to force the RTOS to grant it a higher privilege.

2. Advanced case - This is where the attacker has taken control of the RTOS and is able

to run arbitrary privileged non-secure code. We believe our work is the first to provide some security guarantees on the compromised ECU even when the attacker has control of the RTOS and task code.

3.4.3 Problem Statement

We aim to design a lightweight, predictable defense mechanism for securing the CAN bus that achieves the following:

1. **P1: Prevent Masquerade Attacks.** The attacker, who has infiltrated an ECU's task, tries to masquerade as another task running in the same ECU *or* in another legitimate ECU on the network, e.g., a compromised dashboard entertainment unit may send valid messages to control the engine RPM or apply the brakes. We aim to ensure that a compromised task can, at the most, only send out messages it was designed to generate and transmit, without controlling the actual message content.

2. **P2: Prevent DoS attacks.** The attacker tries to launch a DoS attack by continuously sending messages. We aim to ensure that under no circumstance can a compromised task exceed the max rate at which it was designed to send out messages.

3. **P3: Prevent Snooping.** The attacker tries to read messages intended for other tasks or other ECUs. We aim to remove any control the attacker has over the actual transmission or reception of a message. The attacker should only be able to read messages that were pre-destined for the compromised task. The attacker should also have no mechanism to know *when* another task or ECU sends a message on the bus.

4. **P4: Ensure low latency to allow real-time operation.** Specifically, we aim to ensure that our approach has the smallest, yet predictable, effect on a task's worst-case

execution time and that the task structure does not change.

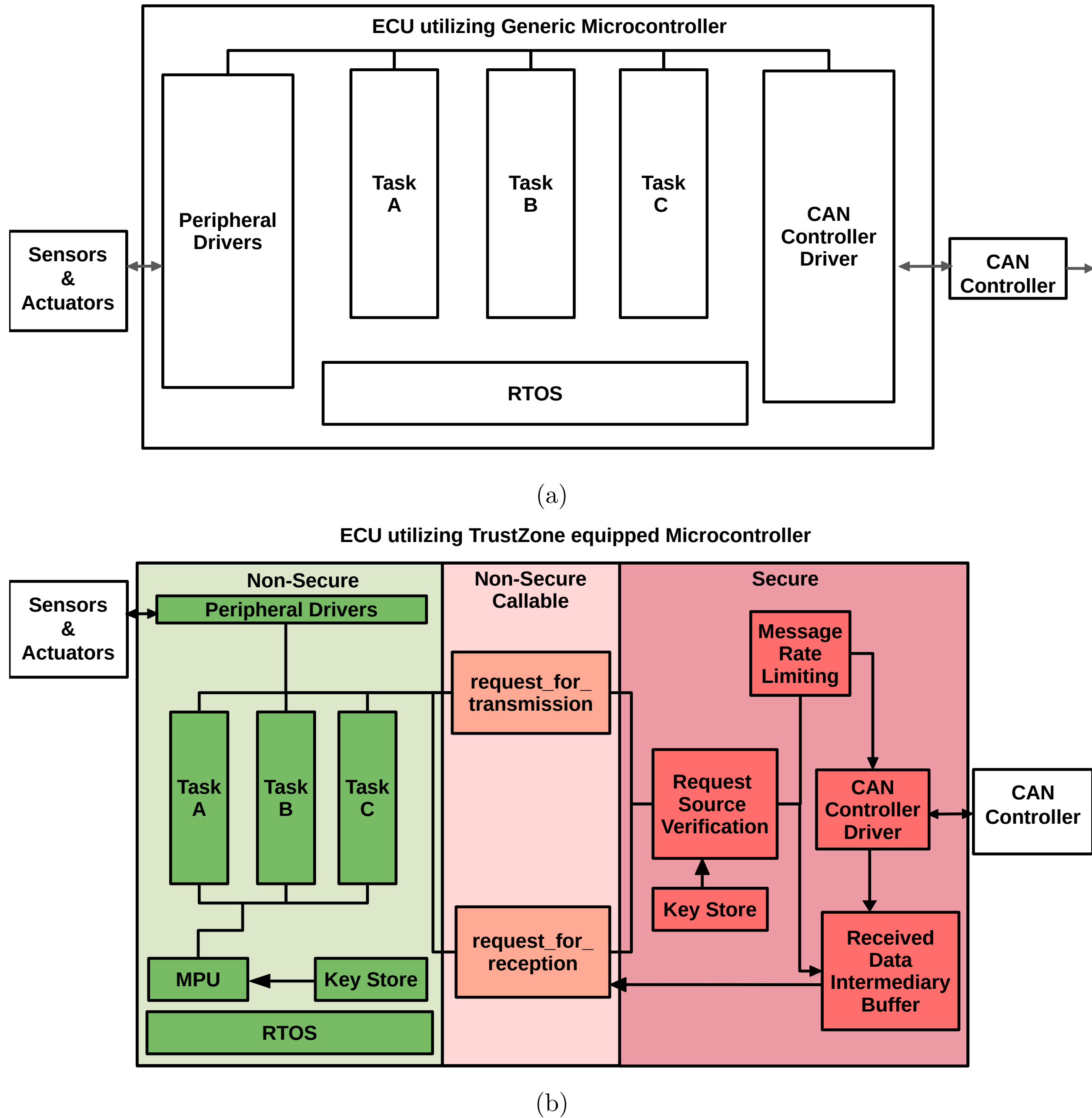
Our proposed technique aims to address each of these problems for the *base case* (Section 3.4.2), and **P2** and **P4** for the *advanced case*. We also partially address **P1** and **P3** for the *advanced case*.

3.5 System Design and Overview

Traditionally, all application code, including peripheral access, executes in unprivileged processor execution mode. Figure 3.2a showcases how such a traditional ECU system might look like. In addition, there is a supervisory code running in privileged execution mode, such as an RTOS, and application code runs in RTOS managed tasks. The RTOS provides scheduling capabilities and task stack management for context switching.

To support TEECheck, modifications must be made to the traditional system setup (Figure 3.2b). Specifically, we consider a TrustZone equipped microcontroller. The RTOS and application code remain in the non-secure domain. We do not need to change the task code structure, other than replacing the CAN driver and transmission code with calls to utilize TEECheck. The task code calls TEECheck's `request_for_transmission` (Section 3.6.1) and

`request_for_reception` (Section 3.6.2) NSC functions, for transmission and accessing received messages, respectively. It must be noted that moving the entire RTOS into the secure region defeats the purpose of using TrustZone, since the secure domain code has unrestricted access to the memory space, only the smallest amount of code should be kept in the secure domain, to reduce the possibility of a vulnerability and, hence, attack surface. Further, by keeping only TEECheck and the CAN controller driver in the secure domain, we force every call to the CAN peripheral to utilize TEECheck.



(a)

(b)

Figure 3.2: (a) Regular ECU system and (b) ECU system utilized for TEECheck

The RTOS is augmented with access to the non-secure domain's MPU which is loaded with task-specific access rights on every context switch. All peripherals (except CAN) necessary for task functionality are partitioned to the non-secure domain. The CAN controller driver

must be paritioned into the secure domain such that it is exclusively accessible to TEECheck. All application code must utilize TEECheck's NSC functions (`request_for_transmission` and `request_for_reception`) to access the CAN bus, as discussed above. While a system timer is required by the RTOS for scheduling tasks, the secure domain requires one more timer. Considering our experimental testbed's (Section 3.7.1) microcontroller has four other timers, we believe this is a reasonable requirement.

It must be noted that the RTOS must be augmented with functionality to manage the per-task secure stack. Since we consider that every task generates CAN messages which are then transferred to and transmitted from the secure domain, stack management must be present for secure domain function calls to allow for safe context switching during task scheduling. Fortunately, most commercial RTOS that we surveyed for our experimental setup, which advertise official support for TrustZone, already provide an extensible mechanism for the RTOS to manage each task's secure stack.

3.6 TEECheck : A TEE based CAN message checker

We now present our TEE based defense mechanism that leverages the new system design (Section 3.5) to address the problems stated in Section 3.4.2. TEECheck is built on two components:

1. **Transmission:** TEECheck uses a two-stage pipeline, one stage for message source verification and another for message frequency enforcement.

2. **Reception:** TEECheck does not allow ECUs direct access to any messages which pass the CAN controller message filtering stage and which are made available for reading. TEECheck verifies the identity of the requesting task before it forwards the message

from the reception buffer.

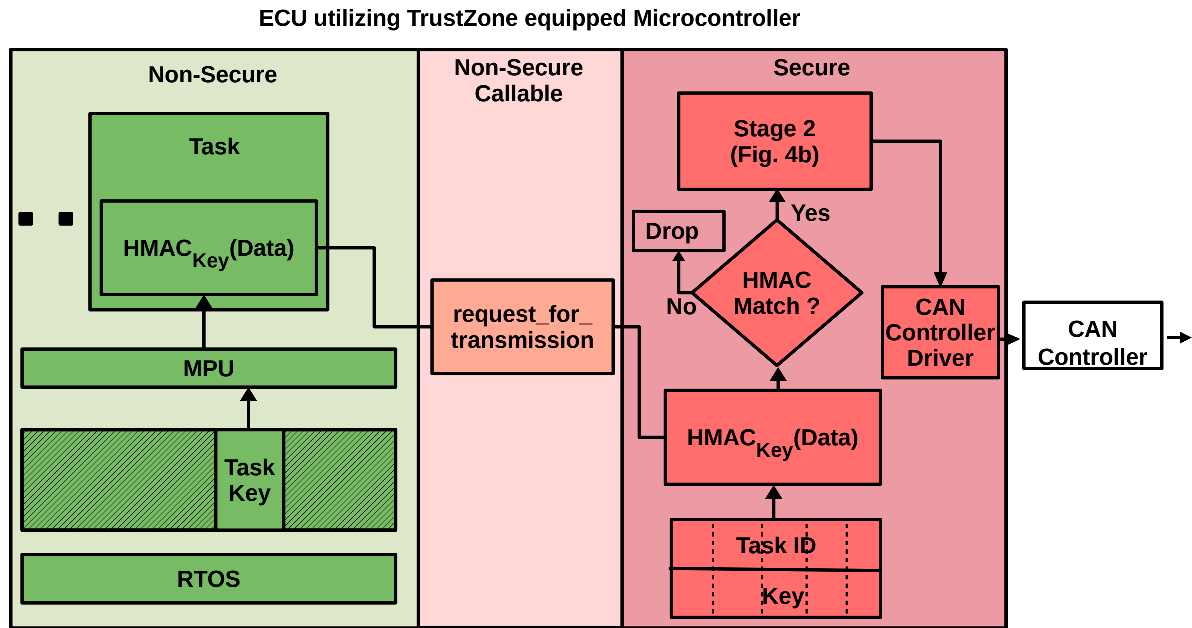
Please note that we address **P4** of our problem statement (Section 3.4.3), by designing TEECheck to work as a set of sequential function calls with no waiting to (1) reduce the impact on a task's WCET and (2) keep the task's structure unchanged.

3.6.1 Transmission

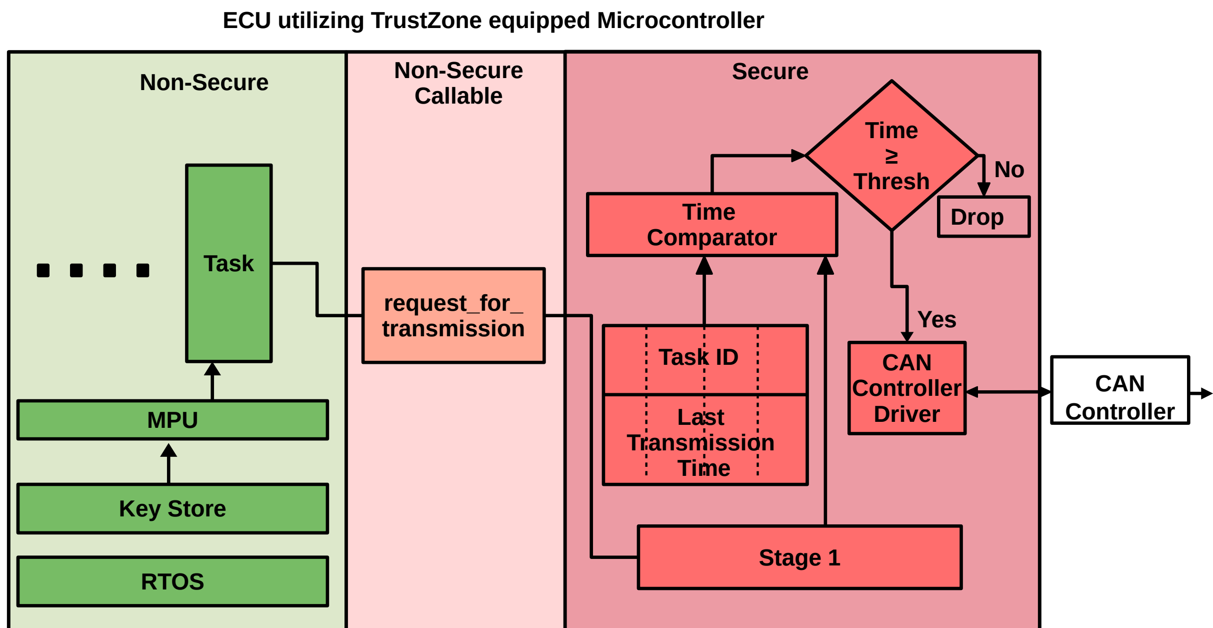
While problem **P3** of our problem statement (Section 3.4.3) is addressed by the reception scheme, the transmission scheme is designed to solve problems **P1** and **P2**. The transmission scheme leverages our proposed system (Section 3.5) to prevent an attacker from masquerading as another task or ECU, as well as from overwhelming the bus (launch a DoS attack). Stage 1 details our message source verification scheme. Stage 2 provides details on limiting the rate of messages that are sent out to the CAN controller.

Stage 1: Source Verification

Stage 1 implements a source verification scheme to eliminate the possibility of a masquerade attack. It utilizes the strict partitioning of the TrustZone to verify the source of a message transmission request. The original system shown in Figure 3.2a, while simple, is flawed from a security perspective. Any misbehaving task can generate messages on behalf of other tasks or even other ECUs in the network. Further, currently available RTOS such as Keil RTX5 (which we use in our experiments in Section 3.7) do not mediate access to device drivers like general purpose OS such as Linux. Rather, the RTOS provides memory management and scheduling capabilities while device access and manipulation (such as access to the CAN controller) is accomplished within the task code. Since the tasks have direct access to the CAN controller, they can simply queue spoofed messages for transmission. However, the



(a) Stage 1



(b) Stage 2

Figure 3.3: (a) Source verification using HMAC (b) Rate limiting messages based on per-task last transmission time

TrustZone provides a strong access control mechanism for peripherals. Shifting peripheral access into the TrustZone while keeping the tasks in the non-secure domain provides us

with the opportunity to control data entry into and out of the secure domain. Under the assumptions of our threat model, the TrustZone interface is the last point at which the attacker still has control over the data. We, thus, propose building a task verification stage at the TrustZone interface before admitting data for transmission on the CAN bus. Every task requesting for transmission must first pass this verification stage on a message-by-message basis or else the message is immediately dropped.

We now discuss the implementation of the verification stage. As noted in the system overview presented in Section 3.5, the RTOS, which controls the MPU, runs as the privileged code while the tasks run in unprivileged mode. Any access to memory locations that are not explicitly marked as accessible by a non-secure task leads to a hard fault generation (a high priority interrupt originating in hardware that halts system execution and requires human intervention to reset the entire system and/or perform cleanup before continuing).

There are two possible avenues of approach to verify a task. One mechanism is by querying the RTOS which task is requesting for CAN access and the other is to build a non-intrusive scheme on top of the RTOS. While utilizing the former approach can be very lightweight, it increases our dependence on the RTOS, enforcing the requirement that the RTOS must not be compromised. However, if an attacker operates under the *advanced case* assumption, such a system is bound to fail immediately. Further, querying the RTOS requires a dialogue between secure domain code and RTOS, which requires RTOS modifications. For automotive systems, this would require expensive re-verification of RTOS functionality and safety. Instead we propose a verification stage between task code and secure domain while keeping the RTOS code untouched. A limitation of the proposed stage 1 is that it still requires certain RTOS guarantees which causes it to partially fail (explained later) when considering an attacker operating under the *advanced case*. However, our proposed approach works well for a *base case* attacker, and being largely RTOS independent, provides a platform for

future work to address all problems for the *advanced case* while still providing performance improvements over related work.

A detailed overview of the steps in the verification stage is presented in Figure 3.3a. We verify the origin of each request by computing and then verifying an HMAC on the message data. HMAC, or hash-based message authentication codes [71], are cryptographic algorithms that take an arbitrary length input and produce a fixed-length output by utilizing a secret key. Only entities possessing the key can generate the same output from a given input, assuming that the HMAC is well designed. We utilize HMACs to verify that a message originates from the correct task. Applying our mechanism to authenticating REMOTE frames, the resultant mechanism would require a minor modification such that it follows the same authentication steps as that in the reception scheme which we detail in Section 3.6.2, where a counter is utilized instead of the message data.

Authentication is done by keeping copies of a table of keys, one key per task, in both secure and non-secure domains. The key table consists of the task identifier and its associated key. An MPU is utilized to restrict a task to access only its designated table entry. We load the MPU during every task context switch with the task-specific address masks such that the task can (i) read its task-specific key-table entry, (ii) read and execute its code section (iii) read and write to its stack in RAM, (iv) call relevant RTOS API, and (v) call TEECheck. Note that, loading the MPU only takes a few clock cycles. The address mask is applied to every memory operation and takes a single cycle due to the MPU being wired-logic hardware. [1]

The key table can be generated and stored in flash during ECU deployment. Alternatively, a small procedure can be added during RTOS initialization to regenerate the keys and store

[1] Loading the MPU is the only guarantee required of the RTOS and we aim to remove this dependency in future work.

a copy in both key tables. An example mechanism of key regeneration could be where the RTOS requests the secure domain code for a fresh set of keys. The secure domain code generates the new keys using a random number generator, saves it to the secure domain table, and forwards it to the RTOS for storage on the non-secure side. Considering the *base case* assumptions, regenerating the keys during RTOS initialization ensures that the MPU is able to hide them before any non-secure task is allowed to run. The task utilizes its key to generate an HMAC based on the data that it needs to transmit. It then calls the `request_for_transmission` function and passes the message pointer, the generated HMAC tag pointer, and the task identifier. Once code execution enters the secure domain, the control is no longer in the hand of the attacker. We read the value of the generated HMAC, look up the key copy from the secure domain key table based on the advertised task identifier, and regenerate the HMAC based on the data to be transmitted in the secure domain. Since we target the CAN bus, the data size is assumed to be 8 bytes. While we do not have any specific requirements regarding the HMAC algorithm, utilizing lightweight algorithms (such as Chaskey [82]) built for small data sizes is advised. Once the task identifier is validated, a CAN message identifier is assigned based on it. In case of the *advanced case* there is only a partial failure of verification stage as the key tables contain identifiers only for tasks meant to run on the ECU. Even if an attacker controls the RTOS, they cannot send out a message which should originate from a different ECU. While we cannot control the actual data, short of recomputing the data in the secure domain, we limit the attacker to only the compromised task and its related message.

Stage 2: Rate Limiting

We introduce stage 2 of the transmission scheme where we rate limit each task's message transmission to address **P2**. Since this stage is entirely within the secure domain, it works

equally well for both threat model cases. The overview of this stage is presented in Figure 3.3b. While utilizing the HMAC scheme in stage 1 prevents a task from sending out false messages on behalf of another task, the attacker could continuously send out valid HMACs to the secure domain to pass stage 1 and force the secure domain to send messages out onto the bus, keeping the bus as busy as possible. While there have been techniques presented in prior work to detect the occurrence of a DoS attack and shut down the offending ECU, these mechanisms are *reactive* in nature and can still disrupt the CAN bus, albeit for short periods of time. We wish to prevent a DoS attack *proactively* before the bus is affected.

Fortunately, since messages are usually generated in a periodic manner inside a vehicle, a system designer is aware of the maximum frequency at which a legitimate task generates messages. We utilize this knowledge to create a per-task rate limiter. Our rate limiting scheme is similar to the more sophisticated mechanism employed in the per-core queue in Carousel [93], to reduce space and computation time. We utilize the hardware timer partitioned to the secure domain to create periodic time *ticks*. The timer counts to the desired period and generates an interrupt. The interrupt handler records the number of ticks. Utilizing interrupts allows for asynchronous operation. While deciding the period value is left to the designer, an example would be to set it to the greatest common divisor of all message periods. For tasks that generate messages of varying frequencies, the worst-case frequency can be used and finding an optimal rate limiter is left for future work. Along with the key table copy, the tick value at the time when the last message was accepted for transmission for the relevant task identifier is also recorded. Stage 2 checks the current tick value, the previous transmission acceptance tick value and the maximum frequency of the task. If the frequency is higher than the maximum allowed, the message is dropped. Else, it is forwarded for transmission.

3.6.2 Reception

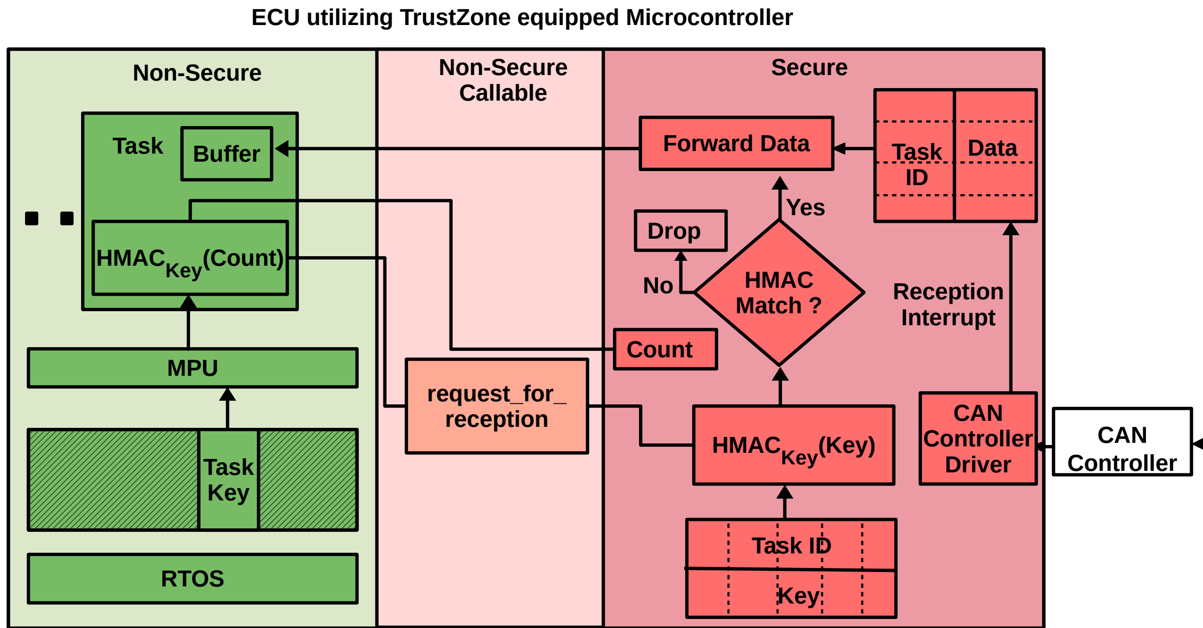


Figure 3.4: TEECheck Reception scheme

The reception scheme (Figure 3.4) addresses **P3** of our problem statement (Section 3.4.3). The reception scheme aims to allow tasks to access *only* those messages with identifiers that belong to them. This prevents a compromised task from snooping for messages to gather sensitive information about the status of the vehicle.

Messages are accepted by the CAN controller when they pass the CAN filtering as described in Section 3.3.1. The filter masks can only be changed by secure domain code since the CAN peripheral is memory-mapped exclusively into secure domain, preventing any non-secure domain code from modifying these masks. This ensures that only messages intended to be received by the ECU, are accepted by the controller. By itself, this limits the attacker to only messages that are to be received by the ECU with zero additional overhead, partially fulfilling the requirements of even the *advanced case*. All messages are handled and stored in a table along with the designated receiving task identifier *asynchronously* via a high priority

interrupt handler. Older values are always overwritten, keeping the contents of the table fresh.

The reception scheme is similar to stage 1 of the transmission scheme. While it would be possible to simply share the task-specific key (made available by the MPU) with the secure domain to ascertain task identity, if an attacker manages to brute-force the correct victim task's key value, the attacker could access *every message* intended for the victim task. Instead, we utilize a per-task counter kept in the secure domain. Every task must first request its counter by supplying its task identifier, then generate the HMAC tag based on this counter and its specific key and then pass the generated tag back to the secure domain for verification using the `request_for_reception` TEECheck NSC function. Regardless of whether task verification passes or fails, the counter is always incremented to prevent an attacker from brute-forcing the correct HMAC tag for a constant input (counter value). Since differentiation between tasks on the ECU requires the MPU, this scheme will fail under the *advanced case* and will be addressed in future work.

Based on the verified task identifier, the corresponding message from the message table is copied to the designated non-secure location pointer passed to the `request_for_reception` after the location is verified as belonging exclusively to the task. For example, an HMAC could be generated from the task key and a nonce during the key refresh phase at system bootup. The task must then write that value to the pointer location passed to the secure domain, which verifies it before overwriting the location with message data.

3.7 Experimentation

We now provide details on our experimental testbed and results. We run different loads on our testbed to gauge the overhead and predictability of utilizing TEECheck. We also apply

Test	Avg (µs)	Med (µs)	Max (µs)	Min (µs)
HMAC-NS (Chaskey)	223	227	229	185
HMAC-S (Chaskey)	247	247	259	242
HMAC-S + RL	254	253	265	248
HMAC-S + RL + CAN	368	357	400	355

Table 3.1: Single task running at highest frequency (RL - Rate Limiting, CAN - CAN controller transmission time)

TEECheck to a well known automotive benchmark as a case study.

3.7.1 Experimental Setup

Our experimental testbed consists of a Nuvoton NuMaker PFM-M2351 development board [2]. It uses Nuvoton's M2351KIAAE implementation of ARM Cortex-M23, based on the ARMv8-M baseline architecture (the least powerful ARMV8-M variant) and has 512 KBytes and 96 KBytes of on-board flash memory and RAM, respectively. The microcontroller supports separate MPUs for secure and non-secure domains, an on-board CAN 2.0b peripheral, and four general purpose hardware timers. We use ARM's Keil RTX5 [77], an automotive functional safety compliant (ISO 26262) RTOS which provides a convenient API for real-time data logging. Our testbed's operating frequency is 12 MHz which allows easy collection of real-time runtime data via the debugger. While the ARM Cortex-R series provides similar operating frequencies as vehicle ECUs (> 100 MHz), we cannot use them as they do not yet support ARM TrustZone.

Since we are not concerned with network-level authentication techniques in this work, we configure the CAN controller to run in Loopback mode, where all transmitted messages are routed back to the controller's reception interface. We enable loading of the MPU in Keil RTX5 for every task context switch. For our experiments, we utilize the ISO standardized Chaskey [82] lightweight HMAC algorithm which is specifically designed for 32-bit micro-

controllers and small data sizes. Please note that we use a software HMAC implementation to show a worst-case overhead for our approach. In the case where a hardware-accelerated HMAC is used, our overhead will further reduce since hardware-accelerated HMAC are known to take less than 100 cycles per computation. Each data point shown in the tables below is generated from 10 experiments due to space limitations for storing timestamps on board in real-time.

3.7.2 Results

We now conduct experiments considering different workloads.

Single task transmitting at highest frequency

The first workload consists of a single task running at the highest possible frequency (constantly looping). This is to simulate a situation where the processor never idles, e.g, an ECU that continuously gathers sensor data and sends it out to a central ECU. We present the results in Table 3.1. Our experimental data shows that when we utilize a secure domain call to generate the HMAC in the secure domain, the entire operation takes the same time as generating the HMAC in the non-secure domain plus the domain switch overhead. The rate limiting stage is very lightweight, only taking an additional 7 us on average. It should be noted that we do not consider reception for these set of results to remove the overhead that could be caused by the CAN reception interrupt. Finally, we also show that the call to the CAN controller for data transmission takes an additional 114 us for a total average of 368 us. Our TEECheck Transmission scheme, thus takes a total of 477 us (223 us + 368 us - 114 us) on average over the base case of providing access to the CAN controller directly to the task. On realistic ECU hardware, our overhead would scale down to 50µs making it extremely

Test	Avg (µs)	Med (µs)	Max (µs)	Min (µs)
Counter request	10	10	11	7
Reception	269	270	271	263
Transmission	373	357	404	356

Table 3.2: Single task running at highest frequency with reception. TEECheck call overhead

lightweight since ECUs send out messages at intervals of milliseconds or higher [70].

Single task transmitting at the highest frequency with reception

We now enable the CAN controller loopback in the same setup as that used in Section 3.7.2 to test the reception overhead. While the interrupt handler runs asynchronously to store the received data in the intermediary data reception table, the TEECheck Reception scheme first checks the task identifier before copying the data to the task's allocated reception space. Results for the reception scheme are presented in Table 3.2. Requesting the counter value takes about 10 us. Results show a 1% difference in transmission overhead from that shown in Table 3.1. As expected, the results for the call to TEECheck's reception scheme are similar to the first stage of the transmission scheme with some additional overhead (15 us) that can be attributed to the additional data copy from the secure domain data reception table to the non-secure domain memory address.

While our transmission overhead is provided for 8 bytes, We also provide results for transmitting different sizes of data, from 1 to 7 bytes in a CAN frame in Table 3.3. Although the HMAC utilizes zero-padding to account for smaller message sizes, the data shows that the overhead for passing the data to the CAN controller is negligible, leading to nearly the same time taken for that when transmitting 8 bytes. Since the behavior of reception and transmission schemes and overhead are so similar, we concentrate on the (heavier) transmission scheme overhead for the rest of this section.

3 tasks - 1 legitimate, 1 attacker and 1 idle

We now simulate ECU with two tasks, one of which has been compromised. A third idle task runs when both of the other two tasks are not ready to run. A priority-based round-robin scheduling mechanism is used which provides equal time to tasks of the same priority and preempts a currently running task if a task with higher priority arrives. All 3 tasks have the same priority. The legitimate task's period is set to 1ms and is always allowed to send a CAN frame. The attacker task does not follow any periodicity and sends out messages whenever it is provided with CPU time. The rate limiting for the attacker task is kept in such a manner that the messages from that task are dropped on every alternate call. Results show that the average, median, maximum and minimum TEECheck call overhead of the legitimate task are **369µs**, **364µs**, **405µs** and **362µs**, respectively. This shows that the average time for calling TEECheck for secure domain processing from the legitimate task remains the same, with a slight increase in the minimum time to account for the overhead due to task switching. In addition, the overhead is negligible even when the attacker has the same priority as a legitimate task and utilizes as much CPU time as possible. Due to the limitations of the Keil RTX5 scheduler, we do not show the results of a higher-priority attacker task since the scheduler would always allow the higher-priority task to run if it is ready to run, impeding the operation of the lower-priority, legitimate task. Modifying the scheduler to deal with such situations is out of the scope of our work. However, regardless of task priority, no task can launch a DoS attack on the CAN bus due to our rate limiting.

4 tasks - 1 emergency, 1 legitimate, 1 attacker and 1 idle

We extend the system setup for the experiment in Section 3.7.2 with an emergency task. We consider a sporadic emergency task, with a minimum inter-arrival time of 3 ms, so as

No. of bytes	Avg (µs)	Med (µs)	Max (µs)	Min (µs)
1	368	256	298	351
2	368	356	398	351
3	369	357	399	352
4	370	357	400	353
5	366	358	398	352
6	367	358	398	354
7	367	358	398	354

Table 3.3: Transmitting different message sizes

to allow easy gathering of the event data while ensuring that the legitimate task meets its deadlines. Results show that the average, median, maximum and minimum TEECheck call overhead of the emergency task are **374µ**s, **370µ**s, **408µ**s and **367µ**s, respectively. There is a slight increase in the overhead by 5 µs from the previous cases. This is to be expected as the emergency task is activated frequently when the other 2 tasks have already entered the secure domain, requiring tear-down of both secure and non-secure domain stacks before switching to the emergency task.

Real world automotive benchmarks

We now augment a well know real-world automotive benchmark [70] with TEECheck. We sort the 9 tasks in a non-increasing order of periods (ranging from 1s to 1ms) and augment each task with a transmission request to TEECheck. Table 3.4 provides our results. Here Augmented Tasks 1 is where only the task with 1 second period calls the TEECheck NSC function and the values are the execution times for that task, Augmented Task 2 is where both 1 s and 0.5 s tasks call TEECheck and the values are presented for the 0.5 s task. Augmented Tasks 9 is when all tasks have a transmission request and we report the execution times for the task with the shortest period. This experiment confirms that TEECheck incurs a predictable overhead on the highest frequency augmented task's execution time even when

scheduled in a round-robin (with same priority) fashion in the presence of other tasks calling TEECheck. When disregarding TEECheck's and CAN controller overheads (477µs and 144µs), the values presented here exceed the actual task execution time by only 30-90 µs. This is likely because TEECheck overhead brings task execution time at-par with the period of the higher frequency tasks, which interfere with another task's execution every time they become ready to run, and due to interrupts from the CAN controller. On realistic ECU hardware, TEECheck's overhead for this benchmark is <60 µs. Please note that we present pessimistic results where every task calls TEECheck as we do not know which tasks, in reality, require CAN access.

3.8 Analyses

We now provide security and real-time analyses.

3.8.1 Real-Time Analysis

We first analyze the real-time properties of our approach in relation to **P4** in our problem statement. Considering our new system architecture presented in Section 3.5, TEECheck simply acts as sequential function calls without additional buffers or other mechanisms that could change the nature of the code flow to affect the task model. As such, TEECheck can simply be modeled as a *constant-time overhead* that must be added to a task's worst-case execution time. Note that this overhead is only applicable to tasks which interact with the CAN bus. The overhead is constant since HMAC generation time on a given length of input data (8 bytes for CAN) is constant and the rate limiting stage is an arithmetic filter that always compares two values: the previous message transmission time and the

Augmented Tasks	Avg (µs)	Median (µs)	Max (µs)	Min (µs)
1	677	692	858	618
2	660	623	776	587
3	1087	1093	1280	1017
4	756	742	895	701
5	978	1007	1092	895
6	1010	1024	1131	912
7	766	766	960	690
8	680	654	874	648
9	666	646	827	615

Table 3.4: Automotive benchmark with increasing number of tasks with TEECheck

current time. Our experimental results based on different system setup scenarios, presented in Section 3.7, verify our analysis. They show almost constant time overhead with minimal variation regardless of system load. The slight variance in data is due to the interrupts being fired by the CAN controller when it receives the looped-back messages. Our approach also shows negligible variation in the case of emergency (sporadic) tasks such as airbag deployment.

3.8.2 Security Analysis

We now evaluate each stage of the transmission and reception schemes for their effectiveness in addressing our problem statement detailed in Section 3.4.3. Stages 1 and 2 of the transmission scheme are designed to address the problems **P1** and **P2**, respectively. The efficacy of stage 1 is based on the assumption of HMAC unforgeability. An attacker could either brute force every HMAC tag value until it matches a valid HMAC tag for the data that it wishes to send, or generate key possibilities for creating the HMAC valid for the data that it wishes to send. Brute-forcing an HMAC algorithm with an n-bit key takes, on average, 2^{n-1} HMAC calculations. Considering an HMAC algorithm (such as Chaskey which we utilized as a part of our experimental setup) which requires a 128 bit key and generates

a 128 bit output tag, the number of tries, on average, for each of the two types of attack would be 2^{127}. While the second attack mechanism is much slower, since the attacker needs to generate the HMAC for every possible key, with a total overhead of about 470 us (223 us for HMAC tag generation in non-secure domain and 247 us for the HMAC tag generation with call to the secure domain), the first attack mechanism would take roughly half the time since the attacker could simply use a counter and pass its value to the secure domain as an HMAC instead of running the HMAC algorithm in the non-secure domain. However, in either case, the verification HMAC generation in the secure domain code cannot be bypassed since accessing the CAN controller must go through TEECheck which always verifies the HMAC first. As such, this makes the attack practically impossible.

Further, the attacker is limited to a short window for guessing the correct HMAC. This is due to the rate limiting stage 2. Since the time-sharing between ECU tasks is enforced by the RTOS, it is guaranteed that the victim task will get to execute. Considering a very basic setup scheduled under the round-robin execution policy, where all tasks have equal priority and context switch is enforced after the same amount of time τ for every task, and considering the time difference δ between messages for the victim task running at highest frequency (that is, a task which is constantly looping over the HMAC generation and call to the `request_for_transmission` NSC function), the victim task (in a system with n tasks) will send out messages every *Msg_Send* which is computed as:

$$Msg_Send = \delta + (n - 1) \cdot \tau \tag{3.1}$$

While an attacker may have prior knowledge of this value, the exact time of the previous transmission during vehicle operation by the victim task cannot be guessed easily, especially because our system prevents tasks from snooping the bus. The attacker must guess during

Msg_Send for successful transmission else even a correct guess will be blocked by stage 2. An intelligent attacker could try to observe execution time differences to detect which stage caused transmission failure. This would require a high resolution timer and can be mitigated by partitioning unused timers to the secure domain. Due to the vastly different execution times of the stages, it is recommended to keep stage 1 and stage 2 in their current positions to prevent an attacker from using the system tick timer (if timer frequency is sufficiently high) to differentiate between stages.

The analysis for the reception scheme is similar to stage 1 of the transmission scheme. The reception scheme is aimed at solving problem **P3** by making it difficult for an attacker to snoop on messages intended for other recipients on the same ECU. Since the attacker has no control over the counter value, the attacker cannot practically perform a replay attack or brute-force the HMAC key, especially if a key refresh occurs on every system reboot.

3.9 Conclusion

We designed a new TEE based architecture for ECUs that effectively partitions the CAN controller from the rest of the ECU code. We presented TEECheck, an on-device TEE based defense mechanism to prevent masquerade attacks, DoS, and information leakage. We implemented our proposed architecture on an actual device, a TrustZone enabled ARM Cortex-M23 based Nuvoton M2351 and showed that our technique has very low overhead (maximum of 494 us and 500 us for transmission and reception respectively), and is very predictable showcasing negligible (around 1%) variance in overhead regardless variations in, and types of, system loads.

Chapter 4

Utilizing hard real-time system predictability to implement control-flow integrity

4.1 Introduction

Real-time systems are now more connected than ever before. Data sharing and interoperation have allowed these systems to better understand their environment and make advanced decisions, improving our quality of life. However, interconnecting such traditionally isolated systems can provide entry-points for attackers to exploit, infiltrate, and modify system operation. For example, the modern car is composed of hundreds of computers, also called electronic control units (ECU), that constantly communicate to maintain smooth and safe vehicle operation. Traditionally, these systems do not communicate with external systems. However, modern conveniences such as Bluetooth connectivity for vehicle occupants, or vehicle-to-everything (V2X), have increasingly exposed these systems to external influence. Prior work [35] shows that such exposure increases the possibility of exploitation by attackers.

At the same time, many such real-time systems are resource-constrained in processing power

and/or memory. For example, vehicle ECUs used for advanced driver assistance systems (ADAS) utilize microcontrollers that operate at hundreds of MHz of clock frequency [72] whereas a modern server or desktop computer's CPU clock runs in the order of a few GHz. Further, ADAS mechanisms such as automatic cruise control, traction control, pedestrian-detection etc., have real-time requirements [25] to ensure occupant safety. Resource limitations and real-time deadline requirements can also be found in other applications such as industrial control systems, robotics, etc. Therefore, both of these attributes must be considered when designing for secure systems.

Once an attacker infiltrates a system, they can perform a variety of system level runtime attacks. We focus on a set of state-of-the-art attacks called control-flow hijacking attacks. Here an attacker either injects code or leverages memory corruption vulnerabilities to overwrite control data, such as return addresses, to manipulate the execution of the program. Return-oriented programming (ROP) [95] is a popular attack that systematically abuses return addresses to *reuse* existing program code. ROP, and its variants are, therefore, also called *code-reuse* attacks. To defend against such attacks, a set of defenses called control-flow integrity (CFI) have been proposed. CFI detect, and prevent, any deviation from the intended control-flow paths of the program. Note that in this work we refer to CFI as "CFI technique" or "CFI mechanism", interchangeably.

Due to the limited resources in embedded systems, modern CFI techniques work around hardware limitations [113] utilize architecture extensions [88],require custom hardware [74], or even reduce detection precision (coarse-grained CFI) and lazily perform checks at checkpoints [23]. Even fewer CFI techniques exist that explicitly consider the timing guarantees of real-time systems. Prior work, such as RECFISH [105] provides a large scale schedulability study of traditional CFI concepts in embedded systems. Here CFI is introduced *in-line* of code-execution, inflating the worst-case execution time (WCET) of tasks. On the other hand,

the authors of [64], similar to previous work integrating general system security in real-time systems [61], introduces CFI budget management techniques to ensure timing guarantees. Here, hardware tracing mechanisms gather control-flow information during program execution, and then verify them at a later time (*out-of-order* to program execution) when the trace buffers are full. Execution budgets are enforced, or CFI verification is temporarily suspended, to ensure that verification does not cause deadline misses in the application.

While these mechanisms trade-off CFI overhead with system performance impact, they may introduce "blind-spots" that could be exploited. By not performing CFI in-line, a smart attacker could modify system output just-in-time before it is sent out. An out-of-order defense mechanism without explicit timing bounds on the completion of verification/enforcement may not be able to detect such an attack in time. In this work, our key observation is that periodic real-time systems, due to their temporal predictability requirements, allow us to a) perform some CFI checks out-of-order of system code execution, b) correctly determine deadlines for these checks so that such blind-spots are mitigated, and c) relax these deadlines, if possible, to improve system efficiency. To determine and relax these deadlines, we were motivated by the fact that many real-time industrial-control and robotics systems gather sensor data (system input) at high frequencies, but send actuator control commands (system output) at lower frequencies. For example, anti-lock braking systems (ABS) sample wheel speed sensors every few milliseconds, but send brake commands at an order of magnitude lower frequency [43]. Combined with the increasing adoption of timing models such as Logical Execution Time [67] model to mitigate data staleness when sharing data between tasks, it is possible to determine when system output is generated, and if there is enough slack to relax CFI deadlines.

Contributions: In this work, we:

1. Propose a new task model for periodic real-time systems that models forward-edge CFI (such as verifying function calls) as separate security tasks.

2. Provide a strategy to push back deadlines of security tasks to increase resource utilization. We provide a schedulability analysis and a correctness study to ensure deadline push backs do not undermine the fact that the control flow integrity policy is strictly enforced before system output is generated.

3. Design and implement a novel CFI framework that utilizes TrustZone for ARMv8-M that:

   (a) Records forward-edge control-flow information, while simultaneously performing backward-edge CFI, via a novel trapping mechanism that also ensures the integrity of CFI records.

   (b) Unlike prior work [61, 64, 105], our framework is the first to explicitly guarantee scheduler integrity.

Note that the focus of our paper is on the real-time task framework to ensure completion of out-of-order CFI policy enforcement, rather than new CFI schemes. As a result, the discussion centers around the newly proposed framework to allow implementation of existing CFI techniques to be performed correctly out-of-order to program execution on commodity ARM Cortex-M hardware. To the best of our knowledge, we are the first to address the real-time perspectives of out-of-order CFI enforcement while assuming a strong adversary capable of compromising the normal world OS.

We shall now look at relevant prior work.

4.2 Related Work

CFI is a widely studied topic in the past couple of decades [31], and a state-of-the-art topic in system security [68], due to the increasing popularity and sophistication of control-flow attacks such as return-oriented programming (ROP [95]) and its variants. CFI mechanisms are built to protect either the *backward-edge*, such as return target of a function, or the *forward-edge* such as the target of a function call using a function pointer, or a combination of backward and forward-edges. While backward-edge CFI is simpler to implement since it requires verifying past control-flow transfers, forward-edge CFI is more difficult to implement since it requires more information to ascertain possible future control-flow paths. Forward-edge CFI, therefore, requires a record of program control-flow information such as a control-flow graph (CFG), which is obtained a-priori to deployment, and is heavily dependent on the quality of the available CFG. Therefore, a wide variety of forward-edge CFI is available, such as fine-grained CFI [31] that provides a correlation between a jump source and all possible jump targets, and coarse-grained CFI [110] that utilizes a less defined CFG and, therefore, allows some reduction in the number of checks performed during control-flow transfers.

Since many legacy or proprietary embedded system software may not have a well-defined CFG available, many CFI mechanisms for such systems, such as Silhouette [113] are designed around coarse-grained CFI. Coarse-grained CFI also allows for interesting relaxations in *when* CFI checks are performed, such as Tyler et.al's control-flow locking [23] where control-flow checks are performed at before and after a function call, i.e., the defense lazily checks for attacks. While it is evident that coarse-grained CFI will have blind spots, which have been exploited by attackers [42], fine-grained CFI too has been successfully defeated [46]. Such attacks can be thwarted by utilizing context-sensitive CFI such as PathArmor [101] that records control-flow transfers and judges each transfer's veracity in context of its neighbours. Unfortunately, PathArmor requires a custom hardware architecture to record control-flow

transfers.

CFI defenses have been increasingly adopted in many higher-end commercial hardware and software systems such as in Microsoft's Windows operating system as part of its Control-Flow Guard defenses [80], the Clang compiler as an optional component [3] as well as Intel's Control-Flow Enforcement [63] technology as part of its more recent line of processors. However, CFI mechanisms built specifically for resource-constrained embedded systems are far fewer in number. This is because such systems are built around less powerful hardware, such as microcontrollers, to manage both cost and SWaP (size, weight and power) requirements and therefore require special software workarounds to support CFI [9, 113], leading to unmanageable overheads depending on the capability of the underlying hardware. Some such as Abad et al. [8] propose dedicated hardware modules to off-load CFI from the processor core. Similarly, TrustZone [109], an architecture extension to support trusted execution in modern ARM processors, has also been used to support CFI [65, 88]. These systems perform all CFI operations in-line with code execution they are forced to simplify the CFI operations to maintain overheads.

In the case of real-time embedded systems, even fewer CFI defenses exist in prior work such as Walls et.al.'s RECFISH [105] which provides an extensive schedulability study of the implementation of common CFI techniques, particularly shadow stacks [30], and labeling to enforce forward-edge CFI [31] on realistic real-time embedded hardware. Shadow stacks protect backward-edges such as function return locations, while labeling is used to correlate a jump source location with possible jump target locations. Bellec et.al [18], on the other hand, propose a detection mechanism that exploits the temporal predictability of real-time systems to detect greater-than-normal execution time but requires specialized hardware. Unfortunately, no currently available defense mechanism considers the temporal relation between *when an attack occurs* (attacker modifies program flow) and the*when the attack takes*

effect (modified program flow affects system output). For example, Bellec et.al's work only detects an attack when the execution time overrun occurs for a segment of code. However, the attacker could have already affected the system by then, such as modified an actuator output and caused damage to the system, rendering the defense ineffective.

Two prior works, Kadar et al. [64] and Hasan et al. [61] are similar to the defense mechanism presented in Chapter 4. Specifically, they abstract CFI, or security mechanisms in general, as security tasks. However, no currently available defense mechanism for real-time systems, including these two works, considers the temporal relation between *when an attack occurs* (attacker modifies program flow) and *when the attack takes effect* (modified program flow affects system output). For example, by not defining such a relation, an attacker could have already affected system output before the verification mechanism detects an attack.

We shall now present the hardware, software and threat models that we consider for the rest of this work.

4.3 Preliminaries

4.3.1 Hardware model

We require that the target system consists of a single ARM TrustZone equipped processor. ARM TrustZone for Cortex-M [109] is a widely available component of the Platform Security Architecture (PSA) extensions on the ARMv8-M state-of-the-art microcontroller architecture. TrustZone creates two processor domains, *non-secure* and *secure*. The former is used to run legacy software, such as an RTOS and task code, whereas trusted supervisory code executes in the latter. ARMv8-M has a flat address space consisting of RAM, flash, and system peripherals. The division of address space between the two domains is enforced via

a vendor hard-wired implementation-defined attribution unit (IDAU) and a system attribution unit (SAU). The SAU supports multiple *regions* depending on the implementation. Each region is a set of registers where secure domain code can programmatically upgrade a non-secure domain address space to secure domain. Switching between the domains has low runtime overhead [81]. Non-secure domain code cannot read/write contents of secure domain memory and can only call specially marked *non-secure callable* (NSC) secure domain code. Other cross-domain memory accesses from the non-secure domain cause a HardFault. We utilize the SAU and the HardFault in our mechanism (Section 4.8). It should be noted that the hard-fault on ARM processors is traditionally utilized to handle critical system exceptions. As part of the ARMv8-M *mainline* architecture (implemented in the Cortex-M33 architecture), an illegal cross-domain memory access (TrustZone violation) launches a dedicated `SecureFault` exception handler. However, we aim to target even the ARMv8-M *baseline* architecture (implemented in the lower-powered Cortex-M23 variant) that bundles a TrustZone violation into a HardFault, and calls the `HardFault` exception handler, to reduce manufacturing costs. We discuss differentiating between a legitimate system fault/exception and an attack in Section 3.8.2. In this work, we refer to either of the fault exceptions as a hard-fault exception for simplicity.

4.3.2 Software Model

The software consists of an RTOS executing periodic real-time tasks with known worst-case execution time (WCET). While the RTOS and application task code normally would execute within the same domain, we split them apart. Specificaly, we execute the RTOS scheduler in the secure domain and task code in the non-secure domain to ensure the integrity of the scheduler and its output (Section 4.8.2). Hardware drivers for peripherals are also kept in the non-secure domain since they are traditionally integrated into task code in embedded

systems. The application program's tasks are referred to as *application tasks*. Application tasks are further divided into *internal* and *output* tasks (Section 4.5.1). The sensor task and actuator task in Figure 4.1 are examples of internal and output tasks, respectively. Separately, we introduce the concept of *security tasks* (Section 4.5.2). We utilize the rate difference between different application tasks to opportunistically relax deadlines of the security tasks (Section 4.6).

We consider that tasks release data at their deadlines, similar to the popular Logical Execution Time model [67] which is widely being considered for automotive and industrial systems since it implicitly addresses the problem of data staleness. While our core idea does not require this assumption, it simplifies our approach and reduces the complexity of our deadline push back model (Section 4.6). It is assumed that data synchronization between tasks happens through shared buffers or similar mechanisms.

4.3.3 Threat model

We assume the use of a system that supports TrustZone. An attacker can compromise any tasks and attain the highest privilege within the non-secure domain. This is representative of real-world threats. For example, task and RTOS code execute at the same privilege level in FreeRTOS [11] by default to eliminate the runtime overhead of privilege level switching. Prior related work, such as RECFISH [105] requires privilege isolation between tasks and the RTOS since they utilize memory protection units (MPU). We assume a *stronger* threat model that considers a compromised non-secure domain.

Similar to other CFI research, we focus on defending against code-reuse attacks, wherein an attacker will target indirect branches (branches on a general purpose register or link register) to implement a forward-edge or backward-edge attack, respectively. Data-only attacks are

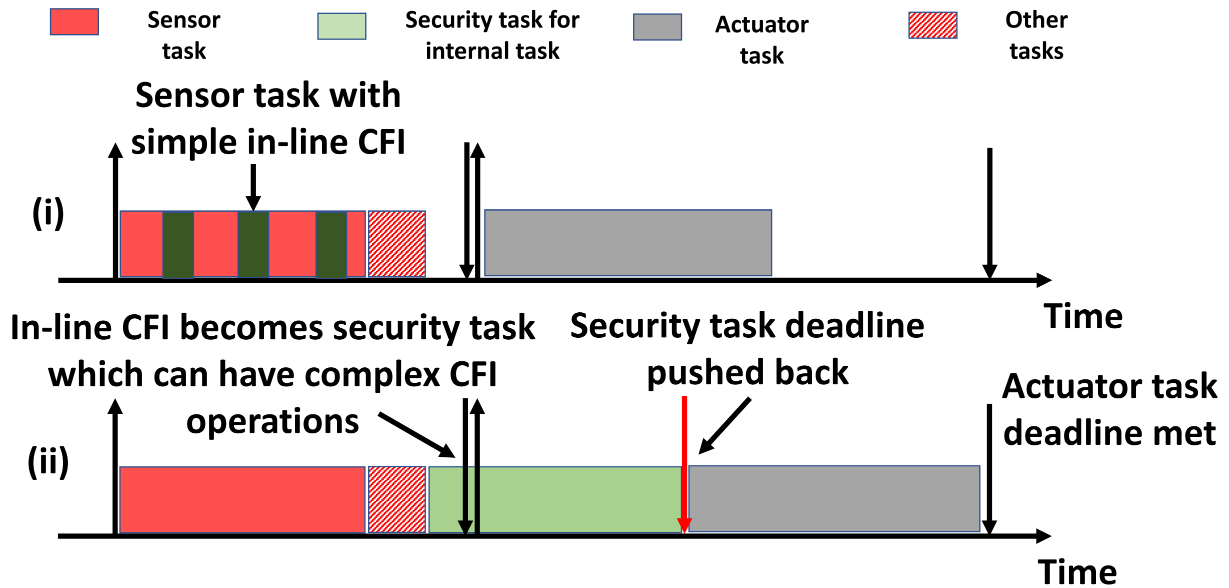


Figure 4.1: (i) Task system where sensor task has simple in-line CFI, (ii) Proposed procrastinating model where in-line CFI is bundled into a security task (Section 4.5.2) which can implement complex CFI operations (accomodate greater WCET) but with pushed back deadlines to lower resource utilization.

out of the scope of this work since they cannot be detected by CFI [62]. Figure 4.2 shows a sample attack: An attacker forces code execution to redirect to an unintended target address by exploiting a vulnerability in `Func A`. By chaining together multiple such branches to different blocks of instructions, an attacker can perform arbitrary computations.

4.4 Overview of Procrastinating CFI

We utilize the periodic nature of real-time tasks to improve schedulability and/or support more complex CFI. A motivating example can be seen in Figure 4.1.(i) which shows a typical control-system where a *sensor task* gathers sensor data, and reports the data after processing to an *actuator task* that sends out commands to control an actuator. The sensor task has in-line CFI, like most prior work. On the other hand, we propose an approach shown in

Figure 4.1.(ii) where the in-line CFI can be consolidated into a separate security task. We utilize system slack to push back deadlines of these tasks, providing an opportunity to utilize complex CFI such as context-sensitive CFI [101]. Our approach consists of the following components:

Procrastinating CFI Task Model (Section 4.5) - Introduces security tasks, differentiates them from application tasks, and provides some important properties of our task model.

Security Task Deadline Relaxation (Section 4.6) - Derives relations to relax security task deadlines such as in applications like ABS [43] where actuation is done at a lower frequency than sensing.

Ensuring Correctness and Schedulability (Section 4.7) - Addresses an implicit relationship between output and security tasks and shows that our model is readily compatible with any resource sharing-aware scheduling algorithm.

Mechanism (Section 4.8) - Presents an overview of our on-device mechanism (Figure 4.3) which logs forward-edge transfers and performs backward-edge CFI while ensuring scheduler integrity.

While our task model itself advances the state-of-the-art by introducing the notion of explicit deadlines to complete CFI to ensure attacker detection before system output is generated, our mechanism further improves the state-of-the-art by extending our model to allow for context-sensitive CFI on commodity hardware.

4.5 Procastinating CFI Task model

Prior work, especially lazy schemes [9, 23] which check for an attack at specific points of time during program execution, cannot always detect an attacker before it affects system

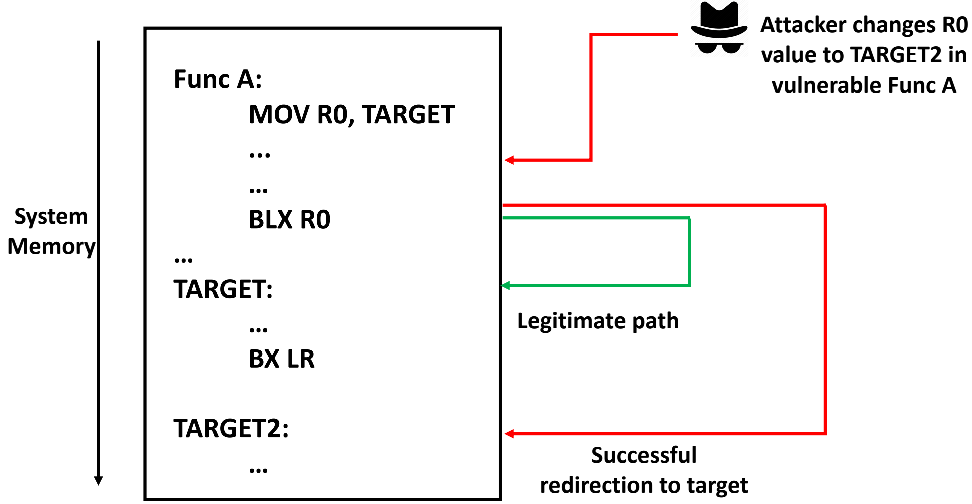


Figure 4.2: Code redirection on a generic microcontroller. Attacker modifies contents of `R0` to change branch target.

output. For example, an attacker could send commands to an actuator controlled by the system before it is detected by the CFI mechanism. We exploit the predictable pattern of periodic real-time systems and design a novel task model that will play a key role in our CFI mechanism. The design of this task model is based on the strict requirement that all pending CFI checks must be completed before the system output is generated i.e., in the example stated above, the CFI check must complete before any command is sent to the actuator. The model also assumes that there exists a mechanism (Section 4.8) to capture information of control-flow transfer events such as the source and target addresses that can be retrieved at a later time for verification.

Notations: We consider that the system consists of n periodic real-time *application tasks* (Section 4.5.1) with known WCET and implicit deadlines. For each application task, we introduce a new periodic real-time *security task* (Section 4.5.2) that models forward-edge CFI that will be performed on the task's control-flow logs. In Section 4.6, we show that these security tasks may not have implicit deadlines. An instance of any of these tasks is called a job. τ_i is the ith application task and $\tau_{s,i}$ is the corresponding security task. D_i, P_i, C_i

are the relative deadline, period and WCET of τ_i, respectively. Similarly $C_{s,i}, D_{s,i}, P_{s,i}$ are the WCET, period and relative deadline of $\tau_{s,i}$, respectively. The jobs $j_{i,k}$ and $j_{s,i,k}$ are the kth instances of τ_i and $\tau_{s,i}$, respectively, and their absolute deadlines are $d_{i,k}$ and $d_{s,i,k}$, respectively. The task utilizations are represented by U_i for τ_i and $U_{s,i}$ for $\tau_{s,i}$. Please see Section 4.5.2 for more details.

4.5.1 Application task model

Our application model is based on a real-time control system, such as an industrial control or robotics system. We consider that the task set is synchronous and, without loss of generality, the tasks are released at time 0. Every task releases its computed data, and consumes input data, only at its deadline and arrival times, respectively, similar to the Logical Execution Time (LET) model [67]. Each application task is either an a) *internal task* that collects data from sensors and/or performs computations, or an b) *output task* that controls actuators or sends messages to external systems. Therefore, the sensor and actuator tasks in Figure 4.1 are examples of an internal and output task, respectively. Our model assumes that the output tasks are given exclusive write access to peripherals (we provide a sample low-overhead mechanism in Section 4.8.2). Since data is released only at deadlines (and in our case, periods), data availability is predictable, regardless of the scheduling algorithm [67]. This removes the need to explicitly consider task dependency. Internal tasks provide data to output tasks either directly or through other internal tasks. As stated in Section 4.3.2, it is assumed that tasks share data via shared buffers or similar mechanisms. We assume that multiple internal jobs can queue data in the shared buffer to be consumed by the output job when it arrives, such as being copied to memory that is accessible only to the output job.

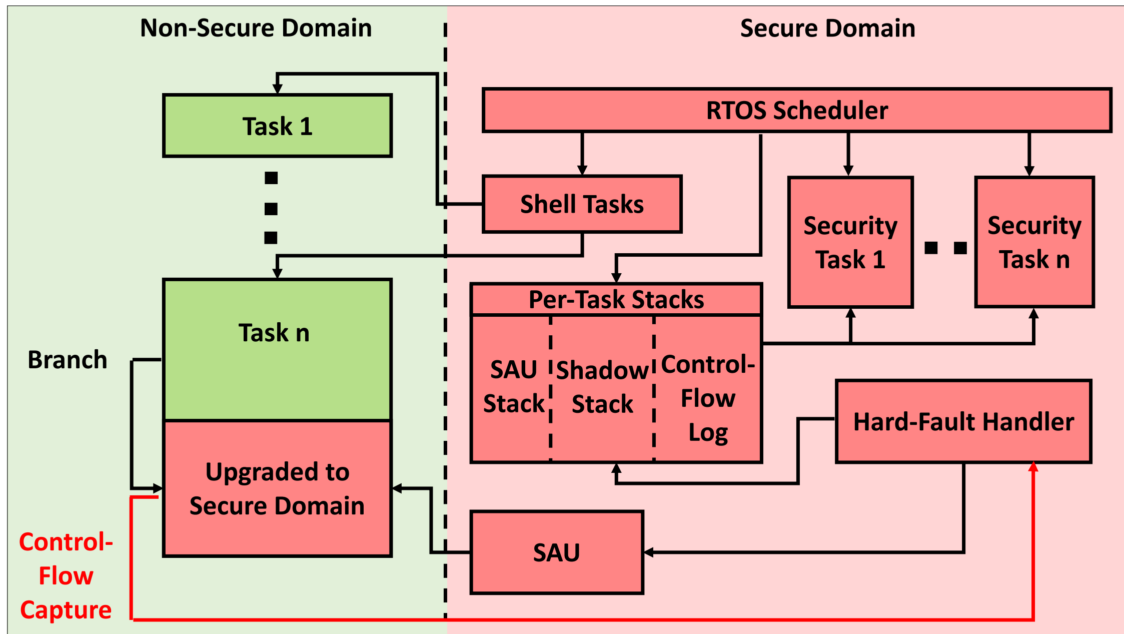


Figure 4.3: Overview of Procrastinating CFI mechanism to capture forward-edge control-flow logs (reading source and destination addresses of branch) and backward-edge verification. Branching into upgraded memory launches fault handler that performs both operations - Section 4.8.

Once the data is consumed by the output job[1], the buffer is purged for newer data.

Data dependency from an adversarial perspective

While logical relations between internal tasks and output tasks can be deduced by generating a directed acyclic graph (DAG) of the flow of data from internal to output tasks, the same relation cannot be assumed to hold true when seen from an adversarial context. For example, consider a system that contains a set of 5 tasks $\tau_a, \tau_b \ldots \tau_e$ with $\tau_a, \ldots \tau_c$ as internal and the remaining as output, such that the data relation between the tasks can be two chains in the DAG graph with $\tau_a \rightarrow \tau_b \rightarrow \tau_d$ and $\tau_c \rightarrow \tau_e$. We cannot assume that an attacker controlling internal task τ_a will only affect the input of τ_b which then filters to τ_d. That is, it could try to

[1]Here an attacker could abuse the temporal assumptions of the framework which we consider in Section 4.7.1.

directly access the shared buffers of both τ_d *and* τ_e unless the system explicitly prevents such modifications. While TrustZone (which we use in our mechanism in Section 4.8.2) could be used for isolating these buffers, due to the limited number of discrete memory regions (up to 8 for our test hardware) that could be managed/upgraded by the SAU, we cannot overtly depend on TrustZone to manage every memory region for complex applications. Scheduling techniques must be utilized to reduce the pressure on a TrustZone-based memory upgrade mechanism. In summary, we cannot assume any relation between the internal and output tasks from a security perspective other than that system output is generated by output tasks at their deadline, and output tasks consume all their input data when they arrive.

4.5.2 Security task model

Under the assumption of a strong attacker, all application tasks are considered equally susceptible to an attack. Possible attack vectors include malicious external input supplied to the task(s) either directly (e.g., sensor input) or through shared data from another task that may trigger a bug and allow an attacker to take control of the task. Therefore, it is necessary to check the control path of all tasks before a system output is generated.

A security task encapsulates all the forward-edge CFI that need to be carried out for a corresponding application task (internal or output). Note that CFI only requires control-flow logs (such as those generated by our mechanism in Section 4.8) and does not access the data shared between application tasks. Therefore, security tasks require that the corresponding application tasks execute and generate these logs before they can perform any verification. We discuss the implementation of the security tasks in Section 4.8.4.

The WCET of a security task $C_{s,i}$, under a worst case number of control-flow transfer events during regular execution $\mathbb{N}$, is:

$$C_{s,i} = cfi(\mathbb{N}), \tag{4.1}$$

where $cfi()$ provides the WCET of the CFI implementation given an input set of control transfer events. Further, since a security task tracks the execution of the corresponding application task, it will also inherit its period, i.e., $P_{s,i} = P_i$.

We will now state some important properties of security jobs based on the observations of the behavior of the application jobs. Here we are considering that the system is originally schedulable, i.e., for a set of n tasks, the total utilization of the tasks satisfies $\sum_{i=1}^{n} U_i \leq U_A^{sched}$, where U_A^{sched} is the schedulable utilization of a scheduling algorithm A.

Lemma 4.1. *The deadline of a security job $\tau_{s,i,k}$, released at the same time as its corresponding application job $\tau_{i,k}$ is:*

$$d_{s,i,k} \geq d_{i,k}. \tag{4.2}$$

For an output job, the corresponding security job's deadline must equal the output job's deadline since LET generates system output at this time. On the other hand, the deadline for an internal job's security job may be greater depending on conditions we discuss later in Section 4.6. Note that since security tasks require control-flow logs before they can perform verification, it is expected that the scheduler tie-breaks to the advantage of the application task if it shares the same deadline (priority) with the corresponding security task. This behavior, however, is an implementation detail and does not change the scheduling algorithm.

From the perspective of an internal job,

Lemma 4.2. *An attacker controlled k^{th} job of an internal task τ_i targeting an upcoming output job $\tau_{j,l}$, will be detected by the security job $\tau_{s,i,k}$ before the system generates its output*

if:

$$d_{s,i,k} \leq d_{j,l}. \tag{4.3}$$

Proof. The LET framework generates system output at the output job's deadline. Therefore, a compromised output job must be verified by this deadline. If the security job satisfies Equation 4.3 the attacker will be detected, in the worst case, at the time the system output is generated under any scheduling algorithm. □

Therefore, the deadline of a security task corresponding to an output task must equal the deadline of the output task while the deadline for those corresponding to internal tasks can be deferred. We now derive the deadline relaxation conditions.

4.6 Security Task Deadline Relaxation

It is potentially worthwhile to relax deadlines of the newly introduced security jobs to improve schedulability of task sets. This is possible for security tasks of internal tasks in the case of task sets where *output tasks execute at lower frequencies than internal tasks.* This condition is fairly common in control systems since output tasks must control actuators, which have a physical limit on how often they can respond to commands and/or due to system requirements. For example, ABS in vehicles have a much higher wheel speed sensor input rate than brake application rate [73].

Consider a LET model of two tasks τ_1 and τ_2, where τ_1 is an internal task, such as a sensor task, and τ_2 is an output task, such as an actuator controller. Let's say that $\tau_{1,m}$ and $\tau_{2,l}$ synchronize at time t which is when the lth job of τ_2 arrives and $t \geq d_{1,m}$. While the synchronization of data needs to follow the LET framework for correct system operation,

the same need not be true for security tasks. For example, if the following is true:

$$t < d_{2,l} - (C_2 + C_{s,2}), \tag{4.4}$$

then there is a non-zero amount of time between the synchronization point and the last time unit at which the internal job's security job must complete to allow enough processor bandwidth to the output job and its security job. Since $d_{2,l} - (C_2 + C_{s,2})$ is evidently before $d_{2,l}$, the deadline of the security job of the internal job can be set to $d_{2,l} - (C_2 + C_{s,2})$ while ensuring our initial guarantee that CFI checks always complete in-time before the system generates output. Figure 4.1.(ii) provides an overview of what we aim to achieve. Therefore, based *solely* on Property 4.2:

Lemma 4.3. *Consider output tasks $\tau_j \in \mathbf{O}$, where $\mathbf{O}$ is the set of all output tasks in the system. For the kth job of a security task corresponding to an internal job $\tau_{i,k}$ that synchronizes with the upcoming l^{th} job of τ_j, $d_{s,i,k}$, that satisfies Property 4.2, can be relaxed such that*

$$d_{s,i,k} = min\{d_{j,l} - (C_j + C_{s,j})\}, \tau_j \in \mathbf{O}. \tag{4.5}$$

Proof. To prove by contradiction, consider $d^{'}_{s,i,k} > d_{s,i,k}$. Also, let $\tau_{s,i,k}$ executes up to its deadline. Further, let a pair of output and corresponding security tasks $\tau_{j,l}$ and $\tau_{s,j,l}$ begin execution at $d^{'}_{s,i,k}$ and let j be the `argmin` satisfying Eq. 4.5. Then $d^{'}_{s,i,k} + C_j + C_{s,j} > d_{j,l}$ which results in a deadline miss and is a contradiction.

□

To generalize Eq. 4.5, we define a *push back value* for each security task. Recall from Section 4.5.1 that an attacker controlled internal task may synchronize with any output task. Therefore, let's consider a pair of internal and output tasks, τ_i and τ_j, respectively.

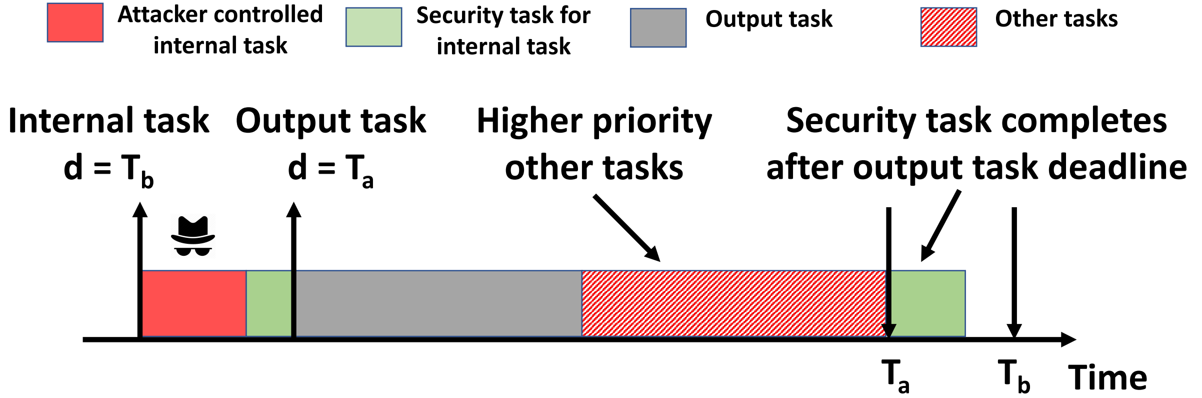


Figure 4.4: Race condition: Attacker is able to execute and affect input of output task. In such a case, the output task's deadline (and output release time under LET) is before the security task is able to complete execution.

For the latest job of τ_i that has a deadline earlier than/at the arrival of lth job of τ_j (and therefore can synchronize with τ_j), the push back value $\Psi_{i,j}^l$ for the job of the security task $\tau_{s,i}$ corresponding to the internal job is:

$$\Psi_{i,j}^l = (l \cdot P_j - (C_j + C_{s,j})) \bmod (P_i). \tag{4.6}$$

However this push back value may not be universally applied to any job of $\tau_{s,i}$. That is, for some instance of $\tau_{s,i}$, applying this push back may violate Property 4.3. Instead, a a minimum $\Psi_{i,j}$, that can be applied for any job of $\tau_{s,i}$, can be derived as:

$$\Psi_{i,j} = min\{\Psi_{i,j}^l\},\ l\ = 1 \dots l^{max}. \tag{4.7}$$

Knowing that τ_i and τ_j are periodic, the push back values will repeat after every hyperperiod

of tasks τ_i and τ_j. Therefore, l^{max} can be stated as:

$$l^{max} = \frac{\text{lcm}(P_i, P_j)}{D_j}, \tag{4.8}$$

where lcm$(\cdot, \cdot)$ returns the least common multiple of its inputs. We now generalize this for $\tau_{s,i}$ with respect to all $\tau_j \in \mathbf{O}$.

Theorem 4.4. *For task sets where output tasks have lower frequency than internal tasks, the security task $\tau_{s,i}$'s deadline can always be pushed back by Ψ_i where*

$$\Psi_i = \min\{\Psi_{i,j}\},\ \forall\ \tau_j\ \in\ \mathbf{O}. \tag{4.9}$$

Finally, the relative deadline of the security task $\tau_{s,i}$ can now be calculated as:

$$D_{s,i} = D_i + \Psi_i. \tag{4.10}$$

4.7 Ensuring Correctness and Schedulability

The deadline relaxation approach described above is only correct if the attacker-controlled task behaves correctly temporally. We now discuss a *race condition* that an attacker-controlled internal task may exploit to break the correctness of the deadline relaxation.

4.7.1 Race condition between output and security tasks

Consider an EDF scheduled system where a security job is released earlier (alongside an attacker-controlled internal job) but with a deadline *greater* than an upcoming output job (Figure 4.4). Under the LET paradigm, this internal job does not synchronize with the

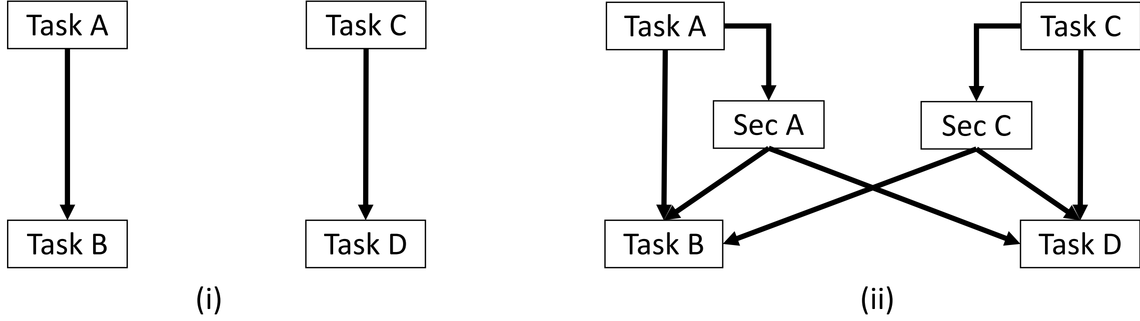


Figure 4.5: (i) Sample data dependency between internal tasks A and C with output tasks B and D. (ii) Dependencies when security tasks are considered.

output job (Section 4.5.1) and hence Property 4.2 needs not hold here. However, an attacker could ignore this behavior and overwrite the data to be synchronized with the output job. Since the security job has a later deadline than the output job, it may be prevented from completing verification until after the deadline of the output job. This could break the correctness of our approach in that we will then not be able to detect an attacker before the system output is generated. This scenario is summarized in Property 4.5.

Theorem 4.5. *If the attacker-controlled internal job $\tau_{i,k}$ and the target output job $\tau_{j,l}$ obey the following relation:*

$$\begin{aligned} a_{i,k} &< a_{j,l} \\ d_{i,k} &> d_{j,l}, \end{aligned} \tag{4.11}$$

then there is no guarantee that the attacker job will be detected before the output of the system is made under EDF.

Proof. The security job $\tau_{s,i,k}$ may have a deadline $d_{s,i,k} \geq d_{i,k}$ (Property 4.1). Consquentially, under EDF, $\tau_{s,i,k}$ may be preempted by $\tau_{j,l}$ until the output job's deadline $d_{j,l}$, and may be unable to complete verification of control-flow events generated by attacker-controlled $\tau_{i,k}$. □

This race condition is not EDF specific and can occur when an output task has a higher priority than a security task of an internal task. We shall now address this problem.

4.7.2 Implicit data-dependency between output and security jobs

The race condition (Section 4.7.1) exists because the independent task model does not explicitly consider data dependency between an output job and every released security job of an internal job[2]. Consider the data-dependency DAG in Figure 4.5(i), which represents a system with 4 tasks such that tasks A and C are internal and task B and D are output tasks. Once security tasks are added to the system, the data-dependency DAG transforms into that shown in Figure 4.5(ii). This is to account for the verification that the security tasks must complete by the time the system output is generated.

Since an internal job and corresponding security job are released at the same time, the scheduler must allow any security job that has been released to be allowed to execute to completion. The task model therefore considers that each security task shares a logical resource with the output task. Therefore, each tuple $(\tau_i, \tau_{s,j}), \tau_i \in \mathbf{O}$ where $\mathbf{O}$ is the set of output tasks, share a logical resource, $\mathbf{R}_{i,j}$ with a security task $\tau_{s,j}$ corresponding to an internal task.

4.7.3 Implications on scheduling

By explicitly defining a shared resource between security and output tasks the race condition can now be prevented by utilizing any shared resource-aware scheduling algorithm such as EDF+SRP [16].

[2] As discussed in Section 4.5.1, from a schedulability perspective, the data dependency can be removed by relying on the LET paradigm

The security job must ”capture” the shared resource when it arrives. If an output job arrives such that it satisfies the race condition (Property 4.5) it is blocked until the pending security jobs ”release” the shared resource. The security jobs ”releases” the shared resource after they have verified all logged control-flow transfer events. The scheduler waits for these ”release events” before scheduling the output job. This solves the race condition.

Other cases that can occur but are implicitly handled by any shared resource-aware scheduling algorithm are:

1. The internal and security job have higher priority than the output job - In this case, the security job will always get completed before the output job gets to execute.

2. The internal and security job arrive later and have a later deadline/lower priority - Under the LET model, the output job consumes its input from any shared buffers at arrival. It will also execute till completion (including preemption by higher priority jobs) before the attacker-controlled internal job gets to execute.

The shared resource design also improves security by reducing the attack surface, since only the security tasks and scheduler communicate via the shared resource and both of these entities execute within the secure domain of TrustZone (Section 4.8.2), which is inaccessible to the attacker under our threat model (Section 4.3.3).

Summarizing our discussion: for a set of application tasks, we have corresponding security tasks. Each security task can be represented by a tuple ($C_{s,i}$, $D_{s,i}$ and $P_{s,i}$), where period $P_{s,i}$ is the same as the corresponding application task's period and $D_{s,i}$ is derived using Property 4.4. Therefore, the total system utilization increases from the original $U = \sum \frac{C_i}{P_i}$, to U_{total} given by:

$$U_{total} = \sum \frac{C_i^{'} + C_{s,i}}{P_i}, \tag{4.12}$$

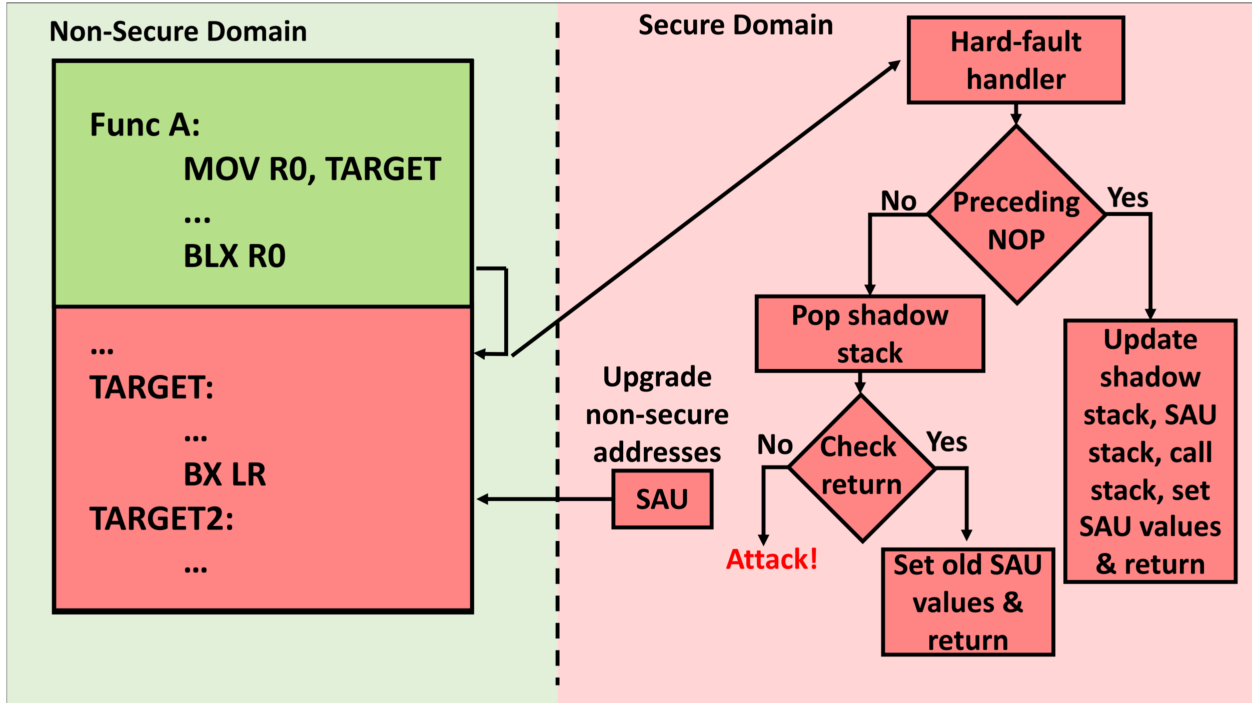


Figure 4.6: SAU based function-call enforcement.

where C_i' includes C_i and the in-line overhead of our mechanism. See Section 4.8.1 for implementation details and Section 4.10 for experimental results.

4.8 Procrastinating CFI mechanism

The task model presented in the Section 4.5 shows that security tasks can utilize system slack to delay execution and perform CFI checks. However, there are two implicit requirements: a) logging of control-flow events, and b) the security tasks must be given an opportunity to execute regardless of whether an attack occurs. For a) we present a novel *function-block* level CFI enforcement that is able to log control-flow data and enforce backward-edge CFI on existing off-the-shelf hardware. A function block includes a function and its prologue and epilogue (compiler-generated additional code). For b) we move the scheduler into the secure

domain.

We utilize TrustZone for ARMv8-M detailed in Section 4.3.1. Note that ”TrustZone” can refer to two different architecture extensions that allows creating a trusted execution environment on ARMv8-A and ARMv8-M processors. TrustZone for ARMv8-M mimics the simplicity of an MPU commonly found in microcontrollers. Unlike in ARMv8-A [83], prior work [81] shows that switching between secure and non-secure domain takes just a few CPU cycles, is highly predictable, and is triggered by a single *secure gateway* (`SG`) instruction. Our mechanism takes advantage of this low overhead and requires minimal, if not zero, modifications to application code.

4.8.1 SAU based function-block enforcement and shadow stack

We now present our approach for bare-metal systems. Since we target resource-constrained systems, we limit our approach to function blocks, reducing the amount (and storage needs) of call logs and in-line overhead. As stated in Section 4.3.3, we assume that the attacker targets indirect branches for forward-edge attacks. Compilers tend to convert function calls to direct branches (corroborated by our case study in Section 4.10.1) since they execute faster, therefore, it would be prudent to check indirect function calls only after they take place. Since we are only aware of attacks where multiple functions are required for an attack [94], indirect calls to locations within the same function can be ignored.

Detecting function blocks in the application binary is difficult since a binary is usually monolithic to optimize space requirements. Prior work uses customized compilers to label functions in the binary. Alternatively, function blocks can be padded with special values such as one of ARM’s many NOP equivalents, or a ”magic” (not an instruction) value, that can be detected at runtime. We perform padding via a command to the linker program that

aligns and pads function blocks with NOPs.

Figure 4.6 provides an overview of our approach. The SAU upgrades all but the currently executing function from the non-secure to secure domain. Since a function call involves a branch to this upgraded memory, it is a cross-domain access that triggers a hard-fault (Section 4.3.1). The handler checks if the target is the start of a function block. If so a) the SAU is updated to shift the non-secure window over this function and b) the old SAU values and return address are stored on separate stacks (shadow stack for return address). The SAU values are calculated at runtime by scanning for block padding to find the start of a function block. A function return is validated by comparing it with the expected return address from the shadow stack. The SAU values are then popped from their stack into the SAU. Any verification failure is detected as an attack.

4.8.2 RTOS modifications

We now apply the scheme in Section 4.8.1 to an RTOS to complete our CFI mechanism (Figure 4.6):

RTOS in secure domain- The RTOS scheduler and timer are moved into the secure domain. Many RTOS, such as FreeRTOS, already maintain an additional per-task stack in the secure domain to save secure function call execution state across context switches. As shown in Figure 4.3, we shift the scheduler into the secure domain, and flip the utility of the task stacks by creating shell tasks that utilize the secure domain stack as the ”main” stack. These shell tasks use the `BLXNS` instruction to the branch start of the task's application code in the non-secure domain. The scheduler, as before, maintains both the non-secure and secure stacks with minimal modifications. Note that on ARMv8-M, there is no performance difference between the two domains and switching between them takes only a few clock

cycles [81]. Therefore, these modifications to the scheduler do not introduce any significant performance penalty.

Implement stack support - SAU values, the control-flow log, and return address shadow stacks are created per task. The hard-fault handler is notified of the currently executing task by the RTOS and updates the correct stacks during runtime.

Implement security tasks in secure domain - These tasks directly accesses the call log stacks.

Other modifications - As stated in Section 4.5.1, we assume that only output tasks can *write* to system output (for example, the UART output buffer). This can be achieved by moving all code that accesses output peripherals into a memory region such that an SAU region can be dedicated to mask this memory (Section 4.3.1). When the scheduler context switches to/from an output task, it can simply downgrade/upgrade the region's security domain, respectively, by flipping a single bit in the SAU. An internal task, therefore, would always cause a HardFault when trying to access this region.

4.8.3 Design Alternatives

A drawback of utilizing a fault handler for CFI is that the hard-fault handler (or `SecureFault` handler - see Section 4.3.1) has the highest priority in the system, and could cause priority inversion. While the fault occurs only at function calls/returns, the number of function calls often depends on the coding style of the developer. Excessive function calls can impose non-trivial overhead in our current design to track the function-level control flow. Fortunately, it is possible to leverage compiler optimizations to reduce the number of function calls. Reducing the number of function calls will also mitigate the degree of path explosion in control flows, at the expense of coarser-grained control. This provides an opportunity to

trade-off performance and security along the dimensions of code size, number of functions, granularity of control flow tracking, as well as run-time overhead. Our experimentation data (Section 4.10) shows that the total overhead of the fault handler is small (our implementation requires a total of $\approx 50-350$ cycles on our test hardware. However, checking for an indirect call takes just ≈ 60 cycles (≈ 5 us). It is possible to further reduce the overhead by only analyzing the transitions that involve safety-critical functions.

While the fault handler overhead is a known limitation that we shall address in the future, using the handler allows greater flexibility to implement complex runtime CFI not possible via binary instrumentation. For systems where this overhead is not acceptable, an instrumentation-based approach can be used. Note that either approach can be used with our task model.

4.8.4 Verification of Control Flow in Security Task

As discussed in Section 4.2 different CFI mechanisms exist in literature that trade-off security and performance: one could, e.g., individually verify the past N control-flow events. PathArmor's [101] path verification performs a depth-first search to find a valid path within the pre-recorded CFG of the application. Security tasks in our framework can straightforwardly implement any of these techniques and our scheduling strategy balances security overhead and schedulability.

4.9 Security analysis

We analyze our mechanism in the context of our threat model (Section 4.3.3). By virtue of TrustZone, we do not have to worry about attacks concerning privilege levels within the non-

secure domain. The attacker can continue to operate at the highest privilege level within the non-secure domain. However, such an attacker will not be able to modify scheduler operation since it cannot cross the TrustZone boundary and directly jump to/modify scheduler code. Also, by moving the scheduler (and timer) into the secure domain, we ensure that a) the timer interrupt will fire and the scheduler will have the opportunity to context-switch to a security task, and b) the scheduler will correctly report the currently running task to the hard-fault handler. Similarly, the attacker cannot access the shadow or SAU stacks, or the function call logs.

Since the hard-fault exception has a higher priority than other peripheral interrupts, its operation is atomic, guaranteeing the integrity of updates to the SAU, shadow, and function call log stacks. Similar guarantees can be afforded to the timer interrupt by allotting it the next lower priority level.

The call logs can be stored in a circular stack to prevent an attacker from overflowing it and the LIFO order allows security tasks to quickly detect an ongoing attack.

Since we do not consider non control-flow attacks, and code-injection attacks can be addressed via the use of mechanisms such as an MPU, we do not upgrade non-secure domain RAM to allow tasks to share data without added complexity.

As discussed in Section 2.1, on the Cortex-M23 architecture, the fault due to a TrustZone violation and the generic system hard-fault are bundled together into the same exception with the same handler that could lead to some confusion to distinguish between an actual system fault and an attacker-influenced TrustZone boundary violation. However, our threat model considers that an attacker would abuse function calls (Branch-with-Link instruction variants) to access secure code. Therefore, it can be possible to distinguish between a fault and an attack by reading the Link register in the fault handler, and determining if it has

been updated due to a branch-with-link instruction. If not, it could be due to a legitimate system fault. Very special cases, such as a jump to secure code due to a fault alongside a simultaneous modification of the link register, are outside the scope of our work and would require application-specific forensic analysis.

Finally, as stated in Section 4.5.2, we consider the worst-case situation where all system tasks, internal or output, require corresponding system jobs. We recommend such a setup to ensure that malicious inputs from the internal tasks do not trickle through intermediary internal tasks and exploit a bug in the output task, regardless of the improbability of such an occurrence. Specific applications may allow for reduced number of security tasks. However, this decision is left solely for the system designer.

4.10 Evaluation

We now evaluate our approach. We present microbenchmarks of our implementation on hardware to compare against prior work, and then evaluate our task model via simulations.

4.10.1 Control flows in Cyber Physical System (CPS) Software

Code used in CPS predominantly follows a predictable control flow sequence. To gain a realistic understanding of the expected overhead of control flow integrity in software programs for CPS, we developed a custom back-end pass in the LLVM compiler suite [73] to count the percentage of indirect control flow transfers in the control loop of widely-deployed real-time CPS, including ArduCopter[4], Rover in ArduPilot[6], PX4[5], and TurtleBot[7]. As shown in Table 4.1, in real-time CPS, forward and backward control-flow transfers are equal or less than just 1% and 6%, respectively, in all programs.

Table 4.1: Control-flow transfers in popular real-time applications.

Software	Forward (% total)	Backwards (% total)	Total
ArduCopter	2139 (0.72%)	7160 (2.41%)	296581
PX4	6211 (0.08%)	44003 (0.61%)	7178749
TurtleBot	647 (0.25%)	1130 (0.44%)	255680
Rover	4 (0.41%)	8 (0.83%)	964

4.10.2 Experimental setup

We utilize a Nuvoton NuMaker PFM-M2351 (ARMv8-M baseline architecture) development board [2] for our experiments. The on-board flash memory and RAM are 512 KBytes and 96 KBytes, respectively. The processor operates at 12 MHz. We modify FreeRTOS according to Section 4.8.2 to support our approach. Modifications include altering the base addresses used by the scheduler to swap the non-secure and secure domain stacks ($\approx$ 20 instructions each for saving and loading the SAU stacks during context-switching), adding 15 lines of C code to initialize and manage the SAU and shadow stacks, and $\approx$ 20 instructions added during RTOS bootup. The majority of our modifications are in the hard-fault handler ($\approx$ 180 lines of C and in-line assembly).

4.10.3 Hardware Overhead

As stated above, we aim to gauge the in-line overhead of our approach. The results are as follows:

RTOS - We modify the FreeRTOS scheduler to store and load the SAU values on a task context switch. We noted overheads of 42 and 45 CPU cycles for these operations, respectively. To lock/unlock output functions as stated in Section 4.8.2 for an output task, $\approx 5-10$ CPU cycles more would be required to flip the relevant SAU region enable bits. The scheduler operation takes a total of 357 cycles on our hardware, including the SAU value load/store operations. Therefore, our modifications add a 28% overhead to scheduler operation. While

Table 4.2: Deadline relaxation of security tasks.

Task set utilization without security tasks	Avg. total util	% of security task with push backs	Avg. % deadline relaxation (95% CI)
0.1	0.14	25.52	3.09 ± 1.47
0.2	0.25	20.85	5.37 ± 2.74
0.3	0.36	22.22	4.84 ± 2.52
0.4	0.49	20.29	4.65 ± 2.44
0.5	0.60	21.54	4.08 ± 2.39
0.6	0.7	23.38	3.71 ± 2.0
0.7	0.82	22.54	2.97 ± 1.73
0.8	0.93	20.29	2.89 ± 1.74

this may seem high, our scheduler overhead is lower than that of the closest prior work (32%), while having a broader threat model (Section 4.3.3) and ability to support more advanced CFI techniques (such as context-sensitivity which requires knowledge of multiple control-flow transfers instead of just one) in the security task while better utilizing system slack. We believe that this overhead is mainly due to the slow on-board memory of our hardware (see below).

Hard-fault handler - 257-326 cycles are required for validating a function call, storing the SAU and shadow stack values, and setting up the new SAU values. 60 cycles are required to determine whether a call occurred due to an indirect branch and log the start and end points. We noted a 53 cycle overhead for function returns.

Comparison with RECFISH

Our approach resembles, temporally, the in-line overhead of RECFISH [105]. However, it exceeds the capability of RECFISH, supporting the possibility of out-of-order CFI checks in security tasks. We report a lower scheduler overhead (28% vs RECFISH's 32%), but a higher function call overhead (386 cycles vs RECFISH's 317). These could be attributed to differences in target architecture. However, both approaches have the similar points of interest, i.e., they operate during a context switch and during an indirect jump. In fact, our

approach incurs an overhead during normal operation *only* when an indirect branch calls a function. Therefore, we believe the schedulability study for RECFISH also applies to our approach and acts as an upper bound. We also believe that with comparable system memory and flash, our approach will have a much lower total overhead.

A note on memory overhead

Our approach requires stacks in the secure and non-secure domain. This is *not* an overhead we introduce. FreeRTOS' default design includes these stacks for supporting TrustZone operation, such as to save context of a secure function call during regular execution. We simply re-purpose these stacks when we move the scheduler into the secure domain.

For each function call, 32 bits are required to store the target address – consumed by the security task – and the return address in the shadow stack. If compiler support exists, a shorter unique identifier can be used per function (similar to most forward-edge CFI [105]). Our case study in Section 4.10.1 shows that indirect calls are very infrequent; the call log stack can therefore be very shallow. Correctly determining the stack size to aid attack detection in case of an overflow during runtime is outside the scope of this work.

Optionally, 32 bits each are required for the SAU start and end limit addresses, which are stored and popped during function calls and returns, respectively. This can be eschewed in severely resource-constrained environments but requires recalculating the boundaries of the function during a return. We assume there is enough memory for such a stack.

4.10.4 Simulation study

We now provide a simulation study of our task model. We a) show that security task deadlines can be relaxed significantly, and b) show that relaxed deadlines improve system

Table 4.3: Percentage of tasksets (/10,000) that are only schedulable under EDF+SRP when push backs are accounted. Original task set utilization does not include security tasks. WCET Ratio is ratio of WCET of security tasks to application task.

WCET Ratio \ Original System Util	0.85	0.9	0.95
0.1	14.35%	61.67%	54.09%
0.2	58.44%	45.69%	27.40%
0.3	37.41%	17.85%	6.77%
0.4	14.15%	4.54%	0.95%
0.5	3.38%	0.64%	0%

schedulability, even under overloaded conditions.

Deadline Relaxation

We evaluate the ability of our approach to relax security tasks deadlines. The results are presented in Table 4.2. The task set utilizations, before the addition of security tasks, are recorded in the first column. Each task set consists of 10 application tasks (2 are output tasks), generated using the UUnifast [22] algorithm implemented under the SchedCAT simulation suite [24]. Output tasks have the longest periods of the task set to allow deadline push backs. We then add security tasks to create task sets of a total of 20 tasks. The WCET of security tasks is calculated as 10% of corresponding application task. We believe this is realistic for forward-edge CFI checks since indirect branches calling functions are usually sparse (RECFISH considers 1 indirect branch for 10^3-10^7 CPU cycles) as corroborated by our case study presented in Section 4.10.1. An average of 20-25% of security tasks have push backs by up to $\approx 8\%$ (minimum of 0.1%) of their deadline showing a clear avenue to defer CFI.

Security task overloading

Since deadlines are pushed back, there is an opportunity to increase security task load (modeling complex CFI operations) and still maintain schedulability. To simulate this, we

generate task sets with utilizations of 0.85, 0.9 and 0.95. We then add security tasks, where the WCET ratio of security task to application task ranges from 0.1 to 0.5, to test how the system behaves under borderline overload conditions. We summarize the results in Table 4.3. we generate 10,000 unique task sets, consisting of 10 application tasks of which two are output tasks, per experiment. Accounting for rounding errors in the utilization of each task set, each data point is the number of task sets that cannot be scheduled by EDF+SRP unless the deadlines are pushed back.

When the original task set utilization is 0.85, when the WCET ratio increases from 0.1 to 0.2, we see an increase in number of tasks sets that are only schedulable when push backs are considered. This is probably because only a few of the tasksets are overloaded and fail the schedulability test without push backs. We then see a declining trend showing the limits of the system from 0.9 and 0.95 system utilization. This shows that it is possible to include more complex CFI and/or other security mechanisms within the security tasks without compromising the schedulability of the system.

4.11 Conclusion

We proposed a new task model that effectively utilizes system slack to defer CFI checks in a bounded manner for periodic, real-time systems. A novel TrustZone-based mechanism was used to record control-flow decisions that can be utilized by any forward-edge CFI mechanism, maintain scheduler integrity, and implement a shadow stack to defend against backward-edge attacks. Hardware microbenchmarks show that our mechanism has in-line overhead similar to the closest related technique, and simulations show that it is possible to not only push back a significant number of security task deadlines but also schedule task sets in overloaded conditions.

Chapter 5

Simulation-based characterization and a priority degradation-based defense against the Mad Monk attack for mixed-criticality systems

5.1 Introduction

Interconnectivity between computing systems is both ubiquitous and a necessity today. From smart phones and tablets, to cars and trains, to even industrial control systems, computers today are interconnected to not only share data, but also improve operation capabilities, efficiency, reliability and safety. A modern cell phone requires connecting to the internet to perform functions that are critical to businesses such as sharing documents or sending emails. Modern cars require internet connection for operations ranging from downloading navigation maps to receiving over-the-air electronic control unit (ECU) firmware updates that improve vehicle performance. Industrial control systems are being increasingly interconnected to visualize plant operations and perform advanced analysis on data gathered in real-time to determine plant operation safety. The exponentially increasing proliferation of data available from such interconnectivity, and reduced cost of processing said data, has

made it viable for the increased utilization of advanced statistical analysis methods such as machine learning and deep learning, allowing system designers to further understand and optimize such systems.

While increased interconnectivity has brought such large quality-of-life improvements for the general public and scientific community, unfortunately, it has also brought along numerous opportunities for misuse and malicious behavior designed to infringe upon user privacy and system safety. As discussed in-depth in Chapter 1, interconnectivity has allowed attackers the ability to gain access to domestic and industrial computing systems. Especially in the latter case, attacks such as Stuxnet [48] and the more recent attack on Ukraine's power grid [33] both have shown that attackers are now able to remotely influence the operation of critical national infrastructure and possibly cause devastating damage to property and life.

With the reduction in cost of computing, and increased interconnectivity, real-time systems too are becoming more complex. While traditionally many of these systems have been simple and isolated from other computing systems, higher safety standards and the need to address software issues that may arise during runtime, many real-time systems now include connectivity hardware and software stacks (ex., vehicle ECUs allowing over-the-air updates) alongside traditional mission critical components. As discussed above, this opens up these traditionally isolated systems to attacks from external malicious entities.

Further, to reduce size weight and power requirements (SWaP) as well as cost of components, many real-time systems integrate hardware and software covering multiple mission goals into the same processing environment. For example, the modern unmanned aerial vehicle, which are used for operations ranging from military applications to reconnaisance, and search and rescue, utilize the same processor for executing the flight stack (ex, PX4 [5]), and image capturing and processing software, etc. Due to the heavy integration of different software components on the same hardware to align with the goal of reducing hardware requirement to

manage SWaP and cost, it becomes necessary to carefully balance the resource usage between each of these various software subsystems, and the safety requirements. To achieve this, a new class of real-time system models have been proposed in the last decade. These types of system models are collectively labelled mixed criticality systems [29]. Mixed criticality systems are designed to delineate between *assuring safety* and *resource efficiency*. They introduce the concept of *safety-critical* tasks, *mission-critical tasks* and other less critical tasks. Safety-critical tasks must be assured against component failure and deemed to be the most critical to maintain the safe operation of the system, while *mission-critical* and other tasks may be deemed to be less critical. Therefore, the system could selectively re-allocate budget if the system showcase abnormal behavior (also called increasing the *system criticality level*) to safety-critical tasks at the expense of other less critical tasks. While mixed criticality systems is still an ongoing research topic, prior work has already shown its applicability to multiple real-world communication mechanisms such as CAN [28] and Ethernet [41].

Therefore, as with any connected system, mixed criticality real-time systems are also vulnerable to attacks such as control-flow attacks discussed in Chapter 4. While such system-level runtime attacks have been widely studied by system security researchers, it is now essential to also consider whether inherent flaws exist within the real-time system models used to analyze such systems, that could be another vulnerable attack surface. To that effect, a recently proposed work by Fisher et al.[1], called the Mad Monk attack, proposes a technique that exploits certain assumptions within the mixed criticality task model ((Section 5.3)) which allows an attacker to have arbitrary control over system criticality level regardless of how critical the attacker controlled code is to the system's safety. This breaks one of the central tents of mixed criticality systems that only the most safety-critical tasks should have

[1] At the time of writing this dissertation, this is currently work-in-progress and portions of this chapter will appear in the published work.

control over the system's criticality level. By controlling the criticality level of the system, the attacker could indirectly control the budget of other tasks in the system since with each criticality level change, the mixed criticality model re-allocates budget from less critical tasks to safety critical tasks to guarantee safe system operation. Mad Monk attack showcases that while mixed criticality systems improves the ability to safely balance the safety assuring with resource utilization efficiency, it can introduce another attack surface for a determined attacker to exploit. We will discuss the specifics of the Mad Monk attack in Section 5.4.2.

With respect to the Mad Monk attack, this work's contributions are as follows:

1. Provide a simulation study of the Mad Monk attack to determine which real-time temporal parameters influence the success (and/or failure) of the attack.
2. A new degradation strategy is proposed that restricts an attacker's capabilities to initiate and sustain the Mad Monk attack over multiple criticality level switches, severely reducing the usefulness of the attack.

We shall now look at relevant prior work.

5.2 Related Work

Many real-time systems cannot be strictly considered as hard or soft since they consist of a combination of tasks that have different degrees of impact on the system if they miss their deadlines. Such a concept of *mixed* task sets consisting of hard and soft real-time tasks, is not new and has been considered in varying degrees in prior literature over the last few decades. Specifically techniques such as bandwidth servers [47, 75] or slack stealing [98, 99] all consider the possibility of re-purposing slack in the system to service soft real-time tasks without affecting the guarantee of scheduling hard real-time tasks.

Mixed criticality systems are a relatively new label attached to such types of systems and could be considered to be a form of generalization of the idea of mixed task systems. *Criticality* of a particular component is it's importance to the safe operation of the system. While *mission critical* components are required to complete the mission objectives of the system, *safety critical* components are required to maintain the safe operation of the system at any given point of time. A common example used is that of a unmanned aerial vehicle (UAV) conducting reconnaisance. Here the subsystems controlling the flight surfaces of the UAV to maintain steady flight (ailerons, motors, etc.) would be considered as safety critical, and the imaging equipment would be considered mission critical. Similarly, the worst-case exeution time (WCET) estimation techniques to model safety and mission critical subsystems (and their tasks) would vary. Safety critical subsystems would, inherently, require more conservative WCET since these subsystems must be more tolerant to component failure, while mission critcial components may be allowed less conservative estimations. However, realistic execution times for safety critical systems may be very different from the heavily conservative WCET estimates. This leads to an interesting problem where realistic estimates differ so much from the conservative estimates, that performing resource allocation and estimation to safely schedule all tasks in the system could involve over-provisioning of resources, leading to increased size, weight, power consumption (SWaP) and system cost. The concept of mixed criticality systems is to allow more relaxed resource requirement estimation for all components of the system considering their operation in more realistic scenarios while allowing re-allocation of resources during runtime from mission critical to safety critical tasks if the system enters a state of operation that is closer to worst-case. That is, the system can be considered to operate at multiple criticality levels and more conservative estimates are required as and when the system achieves a higher state of criticality. The formal concept of considering different execution times for different criticality levels was formally introduced by Vestal [103].

Since Vestal's seminal work, multiple other works in this field have been published. An excellent summary of prior research in this domain is found in this survey by Burns and Davis [29]. For our work, we are concerned with the problem of execution of mixed critical task sets on a single core processor which has been looked at by prior work [15, 56, 89]. Multiple approaches for determing schedulability under fixed priority scheduling, utilizing response time analysis [17, 26] or slack stealing [44]. For EDF based scheduling, new variants of the algorithm such as EDF-VD [76] have been proposed assigns virtual deadlines to increase/decrease the urgency of safety critical/mission critical tasks respectively. have considered Note that mixed criticality task modeling is not just theoretical. Prior work has also determined its applicability to real-world applications such as CAN [28] and Ethernet [41].

While mixed criticality systems are designed to allow for selective degradation (such as by reducing execution or changing priorities) [27, 59, 76, 97] of certain components of the system to ensure safety guarantees, these techniques are presented from the perspective of realism, that is, to increase the applicability of the mixed criticality systems to real systems where degraded components need to be allowed to progress without affecting safety guarantees. However, none of these work consider malicious system manipulation where an attacker *forcefully ensures* system criticality switches to degrade components artificially. A recently proposed work, called Mad Monk (see Chapter 5 for more details regarding the attack) explictly looks at how the system can be forced to degrade certain tasks by employing a low criticality attacker controlled task to increase the execution time of a higher criticality task, causing the system to switch modes. Note that this may appear similar to the problem of resource sharing between criticality levels [111, 112] have been proposed. However, under attacker control, similar behaviors may not be accurately modelled (ex., the attacker task never communicates with high criticality task under regular execution). Further, the attacker could perform actions other than directly communicating with the high criticality task, such

as by performing a cache flush, which may induce a side effect on the high criticality task where it must now spend extra time loading data from memory before it can perform any computation, effectively increasing its execution time and possibly causing the system to switch criticality modes. For reference, there is a wealth of prior work that looks at cache flush attacks, for various purposes such as leaking confidential information [45, 55], and the specifics to perform such attacks on different processor architectures [53, 79]. Note that prior work in mixed criticality systems does look at varying forms of resource separation, that could mitigate such cache-specific side effects, such as a hypervisor [90], however it may not prevent other side effects that have not been considered. Further, patching such issues may be implementation specific and/or require specialized hardware and software.

We shall now formally describe the mixed criticality system task model and behavior.

5.3 Mixed Criticality Task Model

As previously discussed in Section 5.1 and Section 5.2, mixed criticality systems recognize the need to consider different execution time estimates depending on the operational status of the system, especially for safety-critical real-time systems. This is required to balance the need for assurance against failures within the system, and the need to improve resource utilization efficiency. Further, different tasks within the system may have different degrees of importance towards maintaining safe system operation. For example, in a unmanned aerial vehicle (UAV) utilized for search-and-rescue missions, the flight stack that controls the ailerons, motors etc., is a *safety-critical task*, the imaging equipment is a *mission-critical task* and the internal system logging task that logs system status for the operator, is a good to have task. From the perspective of safety, the safety-critical tasks have the highest *criticality* and the good to have tasks are the lowest criticality. Similary, the system's *criticality level*

represents the mode of operation which requires a degree of assurance (and hence execution budget) for safety-critical tasks. Higher the system's criticality level, more the degree of assurance required from such tasks, and more conservative (greater) the worst-case execution budget for safety-critical tasks.

For this work, consider that a system utilizing a single processor, can operate at k different criticality levels or modes. At any given point of time, the system operates at a criticality level γ where $\gamma \in \{1 \dots k\}$. The system always begins operation at $\gamma = 1$ and cannot exceed criticality level $\gamma = k$.

The system executes a task set Γ consisting of n real-time tasks such that $\Gamma = \{\tau_1, \tau_2, \dots \tau_n\}$. These tasks are considered to be of periodic nature for simplicity (we shall consider sporadic execution briefly in later sections). Each instance of a task is called a *job*. Each task $\tau_i, i \in \{1 \dots n\}$ is represented by the tuple $(\overrightarrow{C_i}, D_i, T_i, \chi_i)$. Here, $\overrightarrow{C_i}$ is the set of criticality level dependent execution budgets. At higher criticality levels, safety-critical tasks may perform extra procedures to ensure system safety. It is therefore necessary, with increasing criticality levels, to reclaim budget by degrading, aborting and/or dropping lower criticality tasks and assign reclaimed budget to the higher criticality tasks to maintain safe and schedulable (under an arbitrary scheduling algorithm S) operation since total system budget is a finite resource. We will formally define how execution time for a task, at each criticality level, is determined below. D_i is the relative deadline of the task and T_i is the task period (or minimum interval-arrival time if the task is sporadic). Note that in some prior work [14], with increase in criticality levels, it is expected that the system may generate and handle more events from the higher criticality tasks while other works such as [27] suggest reducing the frequency of lower criticality tasks. Since our threat model requires a high priority low criticality attacker task, assuming a system that increases the period of such a task on a criticality switch, thereby reducing its priority under scheduling algorithms like RM, would

cripple the attacker. Further, as noted in [27] itself, it could be argued that introducing the capability of arbitrarily changing task frequencies during runtime could lead to another attack surface that could be exploited and used maliciously. Therefore, for this work, we consider the simple case that only budgets change with criticality levels, not periods. Finally, χ_i is the criticality of the task. We shall now discuss the specifics of how the execution time is determined at any given point of time.

At any system criticality level γ, for all tasks with $\chi_i \geq \gamma$, a task will have an execution budget of $C_i(\gamma) \mid C_i(\gamma) \in \overrightarrow{C_i}$ where $C_i()$ is a function that provides the budget of a task when it is given the system criticality level, and where $C_i(\gamma) \leq C_(\gamma + 1)$. For all tasks with $\chi_i < \gamma$, jobs of this task can either be dropped or execute with a degraded budget. Burns and Baruah [27] make the case to not drop jobs since it may not be acceptable to system designers that want low criticality jobs to progress even if the system criticality is higher. Therefore, it is expected that the system provides with a degraded (reduced) budget $C_i(\gamma)$ such that $C_i(\gamma - 1) \geq C_i(\gamma)$. This is a similar model utilized in other works such as the proposed variant of EDF to schedule multiple-level mixed-criticality systems, called EDF-VD [76].

Mode changes: Mode changes in mixed criticality systems occur when task τ_i with $\chi_i > \gamma$ has a job that executes past its execution budget at that criticality level, that is, executes without signalling that it has completed for greater than $C_i(\gamma)$. At this point, the budgets of all tasks are adjusted according to their criticality levels.

5.4 The Mad Monk Attack

We shall now discuss the threat model and mechanism of the Mad Monk attack proposed by Fisher et al.

5.4.1 Threat Model

The attacker is in control of a task, or set of tasks, of low criticality but high priority, such that its job can preempt jobs of higher criticality tasks. An example is the communication interface on a drone or unmanned aerial vehicle which may be low criticality due to it not being critical for the safe operation of the drone. Control over such a task(s) could be made possible via a number of mechanisms, such as the control-flow attacks discussed in Chapter 4.

The system is assumed to have the ability to detect violation of temporal parameters of the attacker controlled low criticality task. That is, the attacker cannot violate the execution budget afforded to the task (via some form of a budget enforcement mechanism), as well as violate its own deadline or frequency. That is, the attacker has no control over the scheduling policy and cannot modify any temporal parameters of itself or any other task. This is in line with many prior work in mixed criticality systems that discuss the need for isolation in mixed criticality systems [29]. If any job of the attacker controlled task violates these temporal parameters, it is immediately dropped.

The attacker is also completely or partially aware of the code and data that is utilized during execution of other tasks, particularly a task of the group of tasks which have the highest criticality level within the system. Since our system model considers a single processor system, it is assumed that the attacker has some limited control over the cache which is shared with the high criticality task, allowing it to perform actions such as a cache flush, or some other shared resource (ex: peripheral I/O). It is expected that the system allows some degree of manipulation over one of these shared resources.

Finally, we assume a mixed criticality system where the attacker task τ_a jobs are not dropped, but at the most degraded, when the system criticality level exceeds the criticality level of the attacker task.

5.4.2 Mad Monk Attack mechanism

The Mad Monk Attack, proposed by Fisher et al., is a technique whereby an attacker, in control of any task at any criticality level, is able to force the system to arbitrarily switch criticality modes to a desired criticality level. This, consequentially, allows an attacker to downgrade any victim task τ_v with criticality χ_v as long as $\chi_v < k$ in a k criticality level system.

Consider, for simplicity a single attacker τ_a, and a single victim. The attack is perpetuated in the following manner:

1. The attacker job preempts a high criticality task's job. This task is also called the *intermediary victim* and is denoted by τ_{iv} such that its criticality is greater than the intended victim τ_v, i.e., $\chi_{iv} > \chi_v$. Since the attacker job must be able to preempt the intermediary victim's job, it must also have a higher priority than the intermediary victim job. That is, the Mad Monk attack requires a *higher priority lower criticality* attacker job for operation.

2. The attacker then performs some delaying mechanism to induce an execution *penalty* on the intermediary victim. For example, a cache flush could be an effective technique to induce a delay on the intermediary victim. Cache based attacks have been widely studied for confidential data exfiltration [45, 55] and could also be used by an attacker to flush any shared cache. Once the intermediary victim's job is scheduled to execute, it will have to reload its data into cache which could extend its execution time. Let the maximum penalty an attacker job of τ_a can induce on a job of the intermediary victim task τ_{iv}, at the current criticality level, is $C_{iv}^{pen}(\gamma)$. Note that we assume the penalty to be dependent on the current system criticality level since a task will follow certain code paths, and therefore work with a certain set of data, depending on the criticality

level. Penalty inducing operations, such as a cache flush, will be able to induce delay commensurate to this code and data.

3. Consider that the attacker has sufficiently high frequency such that up to p jobs can induce a penalty. At any starting system criticality level γ, while the worst case execution time at that level for the intermediary victim job is $C_{iv}(\gamma)$, it may actually execute for an average case execution time - $C_{iv}^{avg}(\gamma)$. Therefore, for forcing the system to mode switch to the next criticality level $(\gamma + 1)$, the following equation must hold:

$$C_{iv}^{avg}(\gamma) + p * C_{iv}^{pen} \geq C_{iv}(\gamma) \tag{5.1}$$

The system will switch to criticality level $\gamma + 1$.

4. The attack must now repeat the previous step to gradually increase the criticality of the system to the desired level of $\chi_v + 1$. At this point the criticality of the system is higher than that of the the victim τ_v and the system will automatically degrade (reduce execution budget/modify priority - see Section 5.2), or drop the victim jobs henceforth. The attack is now completed.

Note that the attack requires carefully allowing the intermediary victim to execute so that it can signal to the scheduler that it needs more time than its current criticality budget. This is necessary to allow the attack to propagate. Therefore, the attacker must either yield to the intermediary victim to execute, when appropriate, even if it is higher priority. Similarly, if the attacker controls multiple tasks, it must carefully ensure that each attacker task sequentially executes with the intermediary victim for each attacker induced penalty to take effect.

The advantage of the Mad Monk attack is that it never violates any of the principles of

mixed criticality systems and that it doesn't need to depend on any single penalty induction technique for it to be successful.

Mad Monk is dangerous since it provides a low criticality task the ability to arbitrarily switch system criticality modes with impunity. This, in turn, breaks the central tenet of mixed criticality systems where only the high criticality task has sole control over system criticality mode switches. We shall now perform a suite of simulations to determine how the Mad Monk attack behaves under different system parameters.

5.5 Simulating Mad Monk

We shall now provide simulations of the Mad Monk attack to determine conditions which are most conducive for the attack. The aim is to characterize the attack, allowing us to find a defense strategy which we will detail in Section 5.6.

5.5.1 Simulator setup

We utilize a heavily modified variant of the SimSo [37] simulator to study the Mad Monk attack. The SimSo simulation suite is a event driven simulator which allows us to modify all jobs' execution budgets the moment a criticality change is detected. For the purpose of all simulations presented here, we consider that the system scheduler implements the Earliest Deadline First [32] (EDF) scheduling policy.

For simplicity of analysis, a 3-level criticality system is considered which are termed as (LO,MID,HI). LO mode corresponds to a criticality level (γ) where $\gamma = 1$. Similarly MID and HI are $\gamma = 2$ and $\gamma = 3$, respectively. Attacker tasks will always operate at LO mode - the worst case for the attacker.

A range of tasks, from three to ten are generated, where appropriate. Task sets are generated using the well-known UUnifast-Discard algorithm [22]. All tasks are *periodic.* We shall discuss sporadic test cases explicitly in later experiments. Tasks are first generated disregarding criticality levels, to ensure schedulability, and then a randomized distribution is generated that distributes the task's execution over the three criticality levels. Care is taken that the lowest criticality level has at least 20%-30% of the total execution budget. Every data point that is presented henceforth is based on a simulation run over 100 generated task sets. Further, to bias the system towards being susceptible to an attack, we ensure that the attacker task is the same or higher frequency than the intermediary victim task since we are utilizing the EDF scheduling policy. Higher frequency ensures that more attacker jobs will have earlier deadlines than an intermediary victim job.

Recall that we simulate the penalty induced on the intermediary victim as a percentage of its current criticality level (Section 5.4.2) since it is expected that at a criticality level, the intermediary victim will follow certain code paths and work with a certain set of data which may defer from that at other criticality levels. Therefore, any attack mechanism, such as cache flush, would induce a penalty that is commensurate with the (average) execution time at that level.

Finally, penalties are applied once per attacker job on an intermediary victim job in the simulator. This models the cumulative effect of the attaker job on that intermediary victim job. Of course, subsequent attacker jobs can apply the same penalty (unless a mode change has already taken place). Further, to ensure realistic behavior, once an attacker job induces a penalty on the intermediary victim, subsequent jobs may not apply the same penalty unless the the intermediary victim has had an opportunity to execute. This ensures that the penalty induced by the first attacker job ”takes effect” before subsequent penalties can be applied.

Total Utilization = 0.9, Average execution time = 0.6* Worst-case execution time		
Num Tasks	Attack Penalty	Success %
3	0.3	0
	0.4	0
	0.6	0
	0.9	8
4	0.3	0
	0.4	0
	0.6	0
	0.9	22
5	0.3	0
	0.4	0
	0.6	0
	0.9	22

Table 5.1: Simulation where average execution time is 60% of worst-case for given criticality level. Success % is the number of task sets (out of 100) that show successful attack.

Unless otherwise specified, we shall work with a single attacker and a single intermediary victim. The attacker operates with a $\chi_a = LO$ and the intermediary victim operates at $\chi_{iv} = HI$. The aim is to force some intermediary victim job to cross its MID level budget, effectively causing the system criticality level to switch to HI. At this point, the MID criticality victim task would degrade, signalling a *successful* attack at which point the simulation ends. If the simulator does not end before the hyperperiod (lowest common multiple of all task periods), the attack is considered *unsuccessful* and the simulation is forced to end. All our results show the number of task sets, out of 100 (per simulation run), that show successful attack. We term this results as *success percentage* in the various tables below.

We shall now discuss each simulation run to determine the best (and worst) conditions for the attack

5.5.2 Determining basic conditions with realistic penalty

The aim of this set of experiments is to determine the task set temporal parameters where a realistic penalty of $\sim 30\%$ of the execution time of the intermediary victim, can allow a successful attack to be achieved. To do so we perform experiments on task sets generaed with totally random periods consisting of three, four and five tasks. The task set utilization is kept high ($\sim$90%) since anything lower would defeat the resource utilization efficiency goal of mixed criticality systems. Intuitively it would seem that closer the worst-case execution time estimate, at any given criticality level, with the average execution time of the task's job, would equate to greater probability of success with lower penalties. This is because it reduces the amount of penalty required to force the intermediary victim's job to execute past its current criticality level execution budget (Equation 5.1), ensuring a system criticality switch. Another intuition is that the relative periodicity of the attacker and intermediary victim task would be the dominant trait affecting the possibility of the attack and that the number of tasks has no consequence. This is because Mad Monk relies on the overlap between attacker and intermediary victim jobs for the attack to take place, which is solely governed by the periods of the attacker and intermediary victim tasks. We test these intuitions and arrive at the results summarized in Table 5.1 and Table 5.2.

It is immediately clear that **a) a tighter execution time estimate is required for attack to take place and b) The number of tasks does not seem to have any noticeable effect on the attack success**. The former is advantageous for the attacker since any advancements in the domain of worst-case execution time analysis, which in-turn, improves the utility of mixed criticality systems, increases the probability of attack. The latter provides evidence that strengthens the intuition regarding relative periodicity. We shall see this confirmed in later experimentation too. Note that we tested less than 30% penalty too but saw no successful attack cases. We are led to believe that 30% is

Total Utilization = 0.9, Average execution time = 0.8* Worst-case execution time		
Num Tasks	Attack Penalty	Success %
3	0.3	22
	0.4	18
	0.7	21
4	0.3	14
	0.4	30
	0.7	33
5	0.3	16
	0.4	25
	0.7	21

Table 5.2: Simulation where average execution time is 80% of worst-case for given criticality level. Success % is the number of task sets (out of 100) that show successful attack.

the minimum required for 0.9 total task set utilization, at tight worst-case execution time estimates. Therefore, the table only shows data for 30% penalty and higher.

5.5.3 Changing to harmonic periods

Based on our intuition that the relative periods of attacker and intermediary victim tasks must be playing an important role on the attack, we switch the manner in which we generate task sets. Now, we ensure that the attacker and the intermediary victim task have harmonic periods (i.e., integer multiples of each other), with the attacker task having a higher frequency to bias the system to have a higher attack probability. This also provides us an opportunity to test higher number of tasks per task set cases since harmonic periods significantly reduce the hyperperiod of a task set. Based on the discussion in Section 5.5.2, we keep the total task set utilization as 0.9, and a tight worst-case execution time. A summary of results is provides in Table 5.3. The results clearly show that harmonic periods are, in fact, advantageous, for the attacker, **increasing the successful attack cases by up to 3 times than that shown in the prior section**.

Total Utilization = 0.9, Average execution time = 0.8* Worst-case execution time		
Num Tasks	Attack Penalty	Success %
5	0.3	84
	0.4	83
	0.5	81
6	0.3	89
	0.4	81
	0.5	81
7	0.3	87
	0.4	86
	0.5	82
8	0.3	84
	0.4	83
	0.5	81
9	0.3	89
	0.4	88
	0.5	88
10	0.3	90
	0.4	93
	0.5	94

Table 5.3: Changing to harmonic periods immediately increases the attack success ratio by 3 times. Success % is the number of task sets (out of 100) that show successful attack.

5.5.4 Considering variations in penalty on a newly released intermediate victim job

We now introduce a little more complexity to determine the best case scenario for the attacker. Here we specifically look at how the attack efficacy is affected if the attacker is not able to penalize a *newly released* (released but not executed) intermediary victim job. The reasoning here is that since the intermediary victim job has not executed, it may not have performed initialization routines, including loading data. Therefore performing certain actions, such as a cache flush, may be less effective on the job than after it has had some time to execute.

We test three cases here. The first is if the newly released intermediary job cannot be

Total Utilization = 0.9, Average execution time = 0.8* Worst-case execution time.		
Num Tasks	Attack Penalty	Success %
5	0.3	5
	0.4	11
	0.5	13
6	0.3	12
	0.4	10
	0.5	7
7	0.3	5
	0.4	5
	0.5	5
8	0.3	11
	0.4	8
	0.5	7
9	0.3	10
	0.4	5
	0.5	8
10	0.3	3
	0.4	6
	0.5	8

Table 5.4: Success percentages when attacker cannot induce penalty on newly released intermediary victim jobs. Success % is the number of task sets (out of 100) that show successful attack.

attacked, i.e., if the job has not yet executed, the attacker cannot induce any delay on it. Simulation results are provided in Table 5.4. We clearly see that not being able to attack such a job significantly reduces the number of successful attacks.

To further confirm our observation, we consider a case where the attacker is able to induce up to 10% of the penalty on the newly released intermediary victim job, that it can otherwise induce on a preempted job. Table 5.5. Here it is confirmed that the attacker can rarely succeed if it is only allowed to induce 10% penalty.

Finally, in Table 5.6 we allow the attacker job to induce up to 70% of the penalty it could have induced on a preempted job. We see that at 30% induced penalty on a preempted job, for a newly released job the penalty is $\sim 70\% * 30\% =\sim 21\%$ which is lower than the 30%

Total Utilization = 0.9, Average execution time = 0.8* Worst-case execution time		
Num Tasks	Attack Penalty - Newly released multiplier = 10%	Success %
5	0.3	4
	0.4	10
	0.5	9
6	0.3	8
	0.4	4
	0.5	15
7	0.3	5
	0.4	11
	0.5	8
8	0.3	8
	0.4	6
	0.5	6
9	0.3	10
	0.4	9
	0.5	11
10	0.3	8
	0.4	7
	0.5	5

Table 5.5: Attacker can induce up to 10% of the penalty on the newly released intermediary victim job, that it can otherwise induce on a preempted job. Success % is the number of task sets (out of 100) that show successful attack.

threshold required for successful attacks that we surmised from our results in Section 5.5.2. At 40% induced penalty on a preempted job, for a newly released job, the penalty is $\sim 70\% * 40\% =\sim 28\%$ which is much closer to the threshold and shows a large increase in successful attacks, validating our observations from before.

5.5.5 Converting the attacker task into a sporadic task

Until this point, we have been discussing purely periodic task sets. We shall now discuss the effect on the attack if the attacker task is converted into a sporadic task. This is to both simulate an attacker than cannot control its inter-arrival time, as well as to view any

Total Utilization = 0.9, Average execution time = 0.8* Worst-case execution time		
Num Tasks	Attack Penalty - Newly released multiplier = 70%	Success %
5	0.3	8
	0.4	77
	0.5	81
6	0.3	15
	0.4	85
	0.5	83
7	0.3	5
	0.4	79
	0.5	92
8	0.3	15
	0.4	89
	0.5	85
9	0.3	8
	0.4	88
	0.5	84
10	0.3	5
	0.4	85
	0.5	87

Table 5.6: Attacker can induce up to 70% of the penalty on the newly released intermediary victim job, that it can otherwise induce on a preempted job. Success % is the number of task sets (out of 100) that show successful attack.

behavior that may not be apparent in the periodic case. Intuitively, an attacker controlled sporadic task (without any control over the inter-arrival time), may be less effective than a harmonic period attacker. However, this experiment shows that in some cases, a sporadic attacker task may outperform a periodic attacker.

This set of simulations is set up as a comparison. We first generate a purely periodic task set as before (Section 5.5.1) and run the simulation. We then run the same task set with the attacker task a sporadic task with minimum inter-arrival time the same as the period of the attacker task in the purely periodic task set, and the maximum inter-arrival time up to 3 times this value.

For this simulation we set the penalty to be 30%, the total task set utilization is set as 0.9

Total Utilization = 0.9, Average execution time = 0.8* Worst-case execution time			
Num Tasks	Experiment	Success %	Earliest Successful Attack %
5	Periodic attacker	83	45
	Sporadic attacker	81	0
6	Periodic attacker	87	38
	Sporadic attacker	82	0
7	Periodic attacker	88	46
	Sporadic attacker	81	0
8	Periodic attacker	89	49
	Sporadic attacker	85	0
9	Periodic attacker	87	41
	Sporadic attacker	87	0
10	Periodic attacker	89	58
	Sporadic attacker	88	0

Table 5.7: Comparing success rate of periodic and sporadic attacker for different number of tasks in task set. Success % is the number of task sets (out of 100) that show successful attack. Earliest Successful Attack % is where a task set shows strictly earlier successful attack occurrence under the specific type of attacker.

and a tight WCET (average case execution is ∼0.8* WCET). As before, the attacker and victim have harmonic periods. Based on our discussion in prior sections, these conditions should show a high rate of success for periodic attackers, and should show lower rate of success with sporadic attackers since there is no control over their inter-arrival time.

While Table 5.7 provides some evidence to the contrary, it also confirms some obvious intuitions. The earliest successful attack % details the number of task sets (out of 100) where an attack takes place strictly earlier (faster) if the attacker is of the corresponding type (periodic or sporadic). Task sets that do not show any attack under either type of attacker, or task sets where both achieve successful attack at the same time. We see that the periodic attacker task outperforms the sporadic attacker under this metric, with the sporadic attacker never achieving a faster attack. This is to be expected since a periodic attacker can always predictably perform the attack earlier, it is possible.

On the other hand, we see there is barely any difference in the success percentage between

Num Tasks	Success % - Only sporadic task successful
Total Utilization = 0.9, Average execution time = 0.8* Worst-case execution time	
5	6
6	7
7	9
8	3
9	6
10	4

Table 5.8: Success % where only the sporadic task was able to achieve successful attack

the periodic and sporadic attacker with the periodic attacker just outperforming the sporadic one. This leads us to believe that the sporadic attacker is able to achieve some edge cases where it is able to achieve an attack where the periodic (and harmonic) attacker cannot. We execute the same task sets and note the cases where the sporadic attacker task is the only one that achieves success. Results for this experiment are in Table 5.8 which showcases that up to 9% of the task sets benefit from a sporadic attacker. This is possible due to siutations such as that shown in Figure 5.1. Here the first sporadic attacker job, shown on the timeline, is released much later than its minimum inter-arrival time ("skipping" the previous job which would have been released if the attacker was periodic), such that it is no longer synchronous with the intermediary victim job. Instead it is released earlier than the intermediary victim job, such that it allows two different attacker jobs (the second one is released at the minimum inter-arrival time) to attack the same intermediary victim job. This induces just enough penalty to cause a mode switch and possibly completing the attack.

The sporadic attacker experiments reinforce our intuition that overlaps between attacker and intermediary victim jobs, where the attacker job can preempt the other since its higher priority need to be curtailed. The Mad Monk attack succeeds primarily due to such occurences.

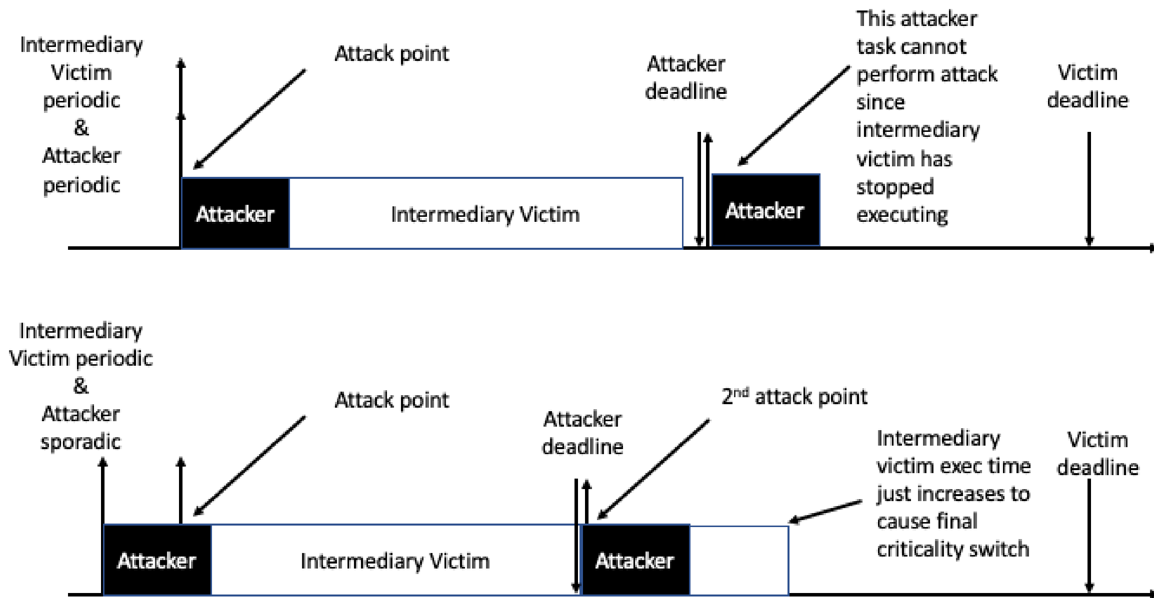


Figure 5.1: Victim here is the intermediary victim job. Periodic jobs are harmonic synchronous. Sporadic attacker job "skips" a previous job, leading to an advantageous condition.

5.5.6 A note on dropping attacker jobs for single and multiple attacker tasks - suicide condition

Up to this point of experimentation, we have considered an attacker job is not dropped (but may be degraded) which allows the Mad Monk attack to propagate. We simulated the situation where an attacker task (or tasks), all operating at LO criticality mode are dropped once system criticality switches to MID.

We term this situation the "suicide" condition since the attacker commits suicide by forcing a mode switch to a criticality level that is higher than the attacker task's criticality. We do not provide a table for this section since under no circumstance, with single or multiple attackers, is the attack successful if suicide is enforced.

This is an expected occurrence due to two design decisions we have taken for the simulator,

which are:

1. Since each attacker job's penalty only takes effect when the intermediary victim job actually executes after the attacker job performs an action, all subsequent attacker jobs (from one or multiple attacker tasks) must wait for this effect to take place. This is especially required if the action that causes such penalty induction is the same for all attacker tasks. Ex: multiple cache flushes before the intermediary victim has had a chance to execute and load memory makes little sense. Therefore, if at any point of time, an attacker job induces just enough penalty to cause the mode switch, other attacker jobs will be dropped and will never have a chance to induce any penalty. Therefore, at any given point of time, only a single attacker job is responsible for a mode change.

2. Our distribution function that divides the execution budget over multiple execution levels ensures that the budget for the LO criticality is the least of the budgets for other criticality levels. This closely mimics real-world mixed critical systems since LO criticality would have the least conservative WCET budgets. Therefore, applying even 90% penalty on this budget of an intermediary victim task's (τ_{iv}) job operating at the LO criticality level is insufficient to cross the $C_{iv}(MID)$. In fact, successive criticality levels will have much larger execution budgets per task, than the previous level, due to increasingly conservative execution time estimates. Therefore, we may generalize this observation and say that a single attacker job may not induce enough penalty to cause multiple mode switches.

Note that if the attacker controls tasks at multiple different criticality levels, it is able to utilize Mad Monk to the upgrade system criticality to the next criticality level of the most critical of the attacker controlled tasks. That is, n mode changes may be induced by n

attacker controlled tasks via Mad Monk if each of the n tasks operate at successively higher criticality levels. However, requiring control so many tasks, at different criticality levels, may make Mad Monk difficult or even infeasible for the attacker. We shall use this observation in the design of our mitigation strategy.

5.5.7 Simulation summary

We shall now summarize the major points discussed based on the results presented above which helps us characterize the attack.

1. Mad Monk attack requires a high priority attacker that can preempt an intermediary victim job. Additionally a high frequency attacker where multiple jobs overlap the intermediary victim is preferable.

2. Tight worst-case execution time estimates is necessary for the attacker to be successful at reasonable penalty levels.

3. Attacker and intermediary victim having harmonic periods improves the probability of the attack.

4. Under harmonic period conditions, being synchronous may be less beneficial for the attacker. The attacker may benefit, in some cases, from some offset between attacker and intermediary victim job release times. However, it still requires that the attacker is higher priority than the victim job.

5. An attacker job may not induce enough penalty to cause multple system criticality mode switches for realistic induced penalties.

6. Attacker committing suicide (dropped) is most effective at halting the attack.

We shall now discuss a mitigation strategy for the Mad Monk attack.

5.6 Criticality level-aware priority degradation to mitigate Mad Monk

We will now design a mitigation strategy that can be implemented to reduce the effectiveness of the Mad Monk attack on mixed criticality systems. Note that the Mad Monk attack is a general exploit within the mixed criticality task model where the attacker may utilize any technique (ex: cache flush), applicable to the particular system, to propagate the attack. Similarly, to ensure adoption, the mitigation strategy must transparently overlay any scheduling policy to mitigate the attack without the need for system-specific defenses.

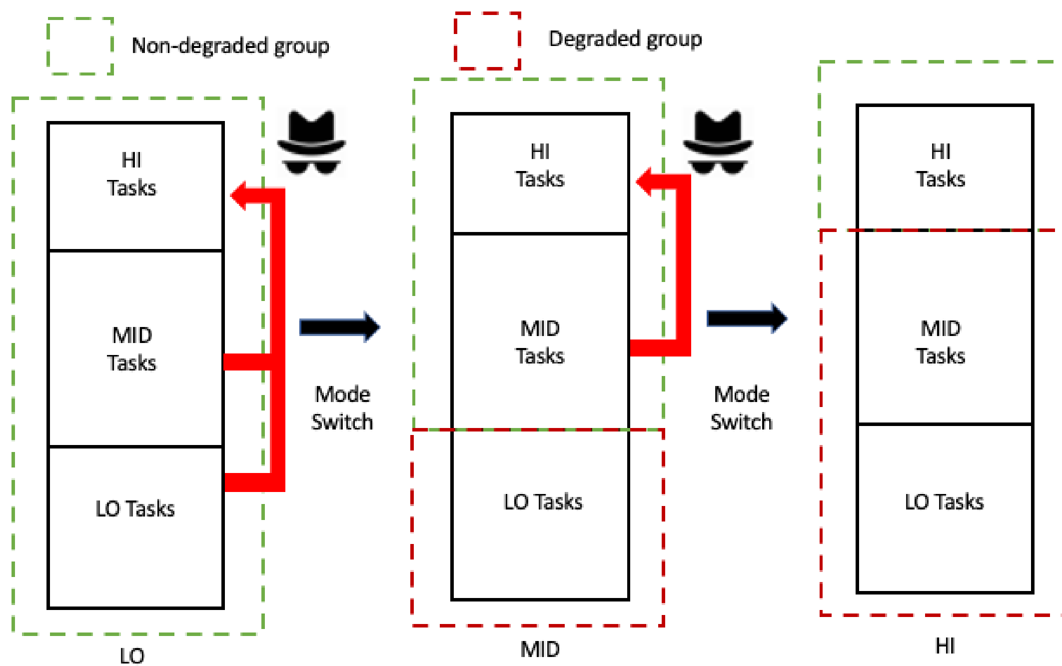


Figure 5.2: Criticality based degradation strategy implemented on a 3 criticality level system. Each criticality change reduces the number of possible attackers that can target a HI criticality intermediary victim task.

To mitigate Mad Monk, applying any form of budget degradation strategies[27, 59, 76, 97] will be insufficient. As summarized in Section 5.5.7, it is clear that a higher priority attacker job that overlaps an intermediary victim job, may launch the Mad Monk attack. Further, depending on the technique utilized to induce penalty, an attacker may be able to perform penalty induction even with limited budget. For example, on x86 architecture, an entire cache line can be flushed with a single `CLFLUSH` instruction which has been used with good effect in other memory modification techniques such as Rowhammer [66]. Therefore, to counter Mad Monk, some form of priority degradation is required to ensure a high priority attacker task is not able to preempt a lower priority higher criticality task. Two approaches may come to mind:

1. Force the requirement to drop low criticality tasks once the system criticality exceeds their level - Our discussion in Section 5.5.6 showcases that this may be the best implicit defense mechanism that can be implemented in the scheduler with the least effort. While dropping low criticality tasks has been discussed in prior work too [27], they conclude that it may be considered impractical for many use cases. For example, a system designer may want certain low criticality tasks to continue execution, albeit in a degraded status, once the system criticality exceeds them. Low criticality tasks such as system logging, especially for forensic analysis at a later time, may require such a design. Completely shutting down logging at higher criticality may make it impossible to ascertain the reasons behind a criticality switch, and/or debug system issues that may arise at higher criticality levels.

2. Utilize a priority modification scheme such as that in Burns and Baruah [27] - Burns and Baruah's work considers a dual criticality system where a lower priority higher criticality task "inherits" the priority of a higher priority lower criticality task once criticality switches from the low to the high criticality level. This is done to prevent

a lower criticality task from ever interfering with the higher criticality task. Since the scheduler cannot assume which task is performing the attack, the scheduler must raise the priority levels of all high criticality tasks to the maximum of lower criticality tasks, to ensure correctness from the perspective of the defense.

Assuming that such a scheme is utilized for a k criticality system, where $k > 2$, this may have an unintended side-effect of priority inversion with tasks with criticality level higher, and priority lower, than the criticality of tasks being degraded. This breaks the spirit of mixed criticality systems since by allowing such priority inversion to occur, we are effectively degrading these mid criticality tasks too when they technically should not be degraded. Further, to maintain schedulability, over-provisioning of resources would be required to ensure these mid criticality tasks meet their deadlines when they have been degraded due to this side effect. This too departs from the central tenet of mixed criticality systems, that is, to improve resource utilization efficiency.

We are therefore, left, with two important considerations to design a mitigation strategy - a) The design must not, via some side effect, penalize tasks whose criticality level is the same or higher than the current system criticality level, and b) must be able to enforce a priority mechanism that effectively mimics the suicide condition discussed in Section 5.5.6, without completely removing the ability for tasks, operating at the same criticality level as the attacker, to progress. We, therefore, propose a *two-level criticality aware degradation scheme* to mitigate Mad Monk, while not violating the principle of mixed criticality systems.

An overview of our approach is presented in Figure 5.2. We introduce the concept of two priority groups, *degraded* and *non-degraded*. The grouping scheme acts as an overlay policy on top of the default scheduling policy of the system. That is, in our approach, the job of any task in the *degraded* priority group are not allowed to interfere with any job (regardless of original priority) of a task in the *non-degraded group*. A degraded group's job must wait for

all non-degraded group jobs that are ready-to-run to first complete before it can be scheduled and allowed to execute. However, within a group, the original scheduling policy determines which job executes first.

Our mechanism operates in the following manner. Figure 5.2 shows a 3 level criticality system. When the system begins execution, it is operating in the LO criticality mode. At this point, all tasks are in the non-degraded priority group. Therefore, as shown by the red arrows, an attacker controlling a LO or MID criticality task can perform Mad Monk on a HI criticality task. Consider that the attacker only controls a LO criticality task for simplicity of explanation. The attacker task will induce a penalty on a HI criticality intermediary victim job to switch the system criticality to MID, at which point all LO tasks (including the attacker task) are added to the degraded group. From a scheduling perspective, the attacker has effectively committed "suicide" since it can never preempt any job of a MID or HI criticality task. On the other hand, since both the HI and MID criticality tasks are in the non-degraded group, they will scheduled based on their priority, eliminating the problem of priority inversion between non-degraded tasks. Therefore, the attacker must now gain control over a MID criticality task to continue the attack, or else the attack has been successfully mitigated. Similarly, once the mode switch occurs to HI criticality, a MID level attacker is also thwarted.

Our degradation technique has multiple advantages over dropping tasks, or modifying task priorities in a manner similar to in [27]. They are:

1. Our approach effectively emulates dropping tasks from the perspective of the Mad Monk attack since a degraded attacker controlled task can no longer preempt (interfere) a non-degraded task. This nullifies the Mad Monk attack which requires a higher priority attacker task to propagate 5.4.2. On the other hand, our approach merely

starves degraded tasks. It still presents an opportunity for these tasks to execute.

2. Our simulations (Section 5.5.6) suggest that for realistic attack conditions, an attacker task at a certain criticality level, may only induce a switch to the next criticality level under suicide conditions. Therefore, our approach, just like the dropping tasks approach requires n attackers at n different criticality levels, for n mode switches. Note that while we do not believe it to be realistic, in case an attacker is able to induce more than one mode switch before committing suicide, our approach does not perform any worse than what could be achieved by the dropping tasks approach.

3. Since we do not modify any arbitrary task's priority on a case-by-case basis, we remove the risk of priority inversion. Therefore, our approach aligns with the principle of mixed criticality systems by not penalizing tasks until the system surpasses their criticality. In fact, by moving lower criticality tasks to the degraded group with each criticality switch, we are reducing the interference seen by tasks in the non-degraded group, effectively improving their response time.

4. Our approach ensures that degraded tasks are *blocked* by non-degraded tasks regardless of priority. However, this group induced blocking, in fact, reduces as criticality changes keep occuring. This is because more tasks will be removed from the non-degraded group and added to the degraded group. They will then be scheduled according to their respective priorities within the group.

5. Since our approach is simply a new degradation strategy, it can be applied in a lightweight manner to any fixed or dynamic priority scheduling algorithm.

While degraded tasks need not be guaranteed against deadline misses (since they are no longer required by the system to maintain safe operation), it may still be useful to showcase the effect of our approach on the worst-case response time of a degraded task. We

shall present a modification to the worst-case response time analysis under a fixed priority scheduling algorithm such as Rate Monotonic (RM) [32] scheduling. Note that in the interest of brevity, we do not present a derivation for the worst-case response time for a degraded task under EDF since multiple approaches have been presented to characterize task worst-case response time under EDF [52, 57]. However, a similar approach as below can be utilized to modify existing analyses.

5.6.1 Response time analysis for degraded group for RM

Consider a k level criticality system with Γ set of tasks, where the current criticality level is $\gamma \in \{1 \dots k\}$, the task group Γ^{γ}_{nd} represents the set of tasks in the non-degraded group, and Γ^{γ}_{d} represents the set of tasks in the degraded group for the current criticality level, such that $\Gamma = \Gamma^{\gamma}_{d} \cup \Gamma^{\gamma}_{nd}$. Therefore, all tasks $\tau_j \in \Gamma^{\gamma}_{nd}$ will block all all tasks $\tau_i \in \Gamma^{\gamma}_{d}$.

Disregarding criticality levels and groups, the worst case response time W_i for a task τ_i is given by iteratively solving the following equation until it converges:

$$W_i = C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{W_i}{T_j} \right\rceil C_j \tag{5.2}$$

Where C_i is the worst-case execution time, and $hp(i)$ is the set of all tasks with priority higher than τ_i under RM. When considering criticality levels, at a current criticality level γ and under our priority degradation strategy, a task $\tau_i \in \Gamma^{\gamma}_{d}$ will have a corresponding worst-case response time W_i, given by:

$$W_i = C_i(\gamma) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{W_i}{T_j} \right\rceil C_j + \sum_{\tau_l \in B^{\gamma}_{i}} \left\lceil \frac{W_i}{T_l} \right\rceil C_l \tag{5.3}$$

Where $B\gamma_l$ is the set of all tasks in Γ^{γ}_{nd} which are lower-priority (set $lp(i)$) than τ_i but will still block it due to the priority grouping. That is, the blocking group $B\gamma_l$ is defined as $B\gamma_l = lp(i) \cap \Gamma^{\gamma}_{nd}$.

Note that with increasing criticality levels, the number of tasks in Γ^{γ}_{nd} will keep decreasing since more tasks will be added to the degraded group. That is, $\Gamma^{\gamma}_{nd} \geq \Gamma^{\gamma+1}_{nd}$. Therefore, the worst-case response times, and hence worst-case deadline misses, for any degraded task will always be at the criticality level when it is first added to the degraded group. Subsequent criticality changes will reduce the number of tasks in the non-degraded group and consequentially reduce the number of lower priority tasks that could block this degraded task. We shall now provide some simulations to showcase the impact of our approach on deadlines.

Total Utilization = 0.9, Average execution time = 0.8* Worst-case execution time		
Num Tasks	Attack Penalty	Deadline Miss %
5	0.3	23
	0.4	15
	0.5	24
6	0.3	19
	0.4	18
	0.5	26
7	0.3	36
	0.4	26
	0.5	26
8	0.3	26
	0.4	27
	0.5	22
9	0.3	21
	0.4	26
	0.5	29
10	0.3	31
	0.4	27
	0.5	23

Table 5.9: Simulating criticality aware priority degradation on LO criticality attacker all other tasks MID or HI criticality. Deadline Miss % is the number of task sets (out of 100) where attacker task misses its deadline.

5.6.2 Simulating effect on deadline misses for attacker tasks

We re-run our simulation setup from Section 5.5. We simulate the worst-case for the attacker, where it is a LO criticality tasks and all other tasks are higher criticality. Therefore, it will be blocked or preempted by all other tasks after the first mode switch. We run the simulations for realistic attacker penalties, with tight WCET estimation (average case execution time = ~ 0.8*WCET), a high task set utilization of 0.9 (i.e., taskset schedulable under EDF). The results are presented in Table 5.9. We see that even with the degradation strategy enabled, only about $20\% - 30\%$ task sets show deadline misses showing that our approach does not introduce too much pessimism. Note that just like in the suicide case where attacker jobs are dropped, our priority degradation approach also does not show any successful attack (where the system reaches the HI criticality mode) once enabled in the simulator.

5.7 Conclusion

In this work we have studied the Mad Monk attack by using a custom simulator. Our sensitivity analyses of the attack showcases the system conditions which are required for the attack to occur. Our observations show that there is a need to introduce a new degradation strategy that prevents an attacker job from preempting a lower priority higher criticality job and performing some action that induces a system criticality switch, among other factors. We present a new 2-level priority degradation strategy that is as effective at thwarting the attack as dropping the attacker task completely, while still allowing the system to schedule low criticality tasks, when possible, allowing them to progress while preventing further propogation of Mad Monk. Our simulations show that the new priority degradation strategy does not introduce a large amount of pessimism for the degraded tasks, while leaving the schedulability of non-degraded higher criticality tasks untouched.

Chapter 6

Conclusion and Future Work

6.1 Summary and Conclusion

In this disseration we have discussed defense mechanisms, each for different system vulnerabilities, for real-time systems. This dissertation highlights the ability to utilize real-time scheduling theory to not only reduce the impact of introducing general-purpose system security defense mechanism in real-time systems, but also improve the ability and correctness of defenses and real-time models, respectively, from a security perspective. In chapter 3, we show that careful system design, that utilizes TrustZone on low-end ARMv8-M microcontrollers, while modeling CAN bus communications as a real-time system, can address system vulnerabilities in a lightweight, predictable manner. In chapter 4, we show that the predictability of real-time system input and output can be exploited to reduce the cost of introducing modern system security techniques such as CFI without violating the correctness of the defense. Finally in chapter 5, we are able to experimentally determine conditions of operation of the Mad Monk attack that exploits an inherent vulnerability in mixed criticality systems, and propose a defense strategy that mitigates this vulnerability within the mixed criticality model. The author of this dissertation hopes that through this work, the real-time systems community receives sufficient additional motivation to apply real-time scheduling theory to improve the integration of general-purpose system security techniques in real-time systems, as well as study real-time scheduling theory from an adversarial perspective.

We shall now discuss how the ideas and techniques discussed in this work can be extended to one other domain within real-time systems.

6.2 Future Work - Vulnerability in real-time systems due to self-suspension

Self-suspension is a widely studied phenomenon in real-time systems over the past decade. Researchers aim to explicitly consider the scenario where real-time computing tasks, especially in resource-constrained system, instead of waiting for the system to respond to certain requests such as retrieving data from memory or servicing an I/O request, may self-suspend and allow other lower priority tasks to continue progressing. While this is clearly beneficial to the operation of the system, the additional scheduling decision point that is introduced due to task self-suspension has proven to be difficult to characterize. A large body of work has been proposed to correctly determine the impact of self-suspension on real-time system guarantees.

The vast majority of hard real-time scheduling theory essentially consists of two parts, a task model specific schedulability analysis and a usually much simpler scheduling algorithm. While the schedulability analyses considers specific task structures and behavior, schedulers themselves have to be implemented with the least number of steps to reduce scheduling overhead during actual runtime. Therefore, schedulers disregard the specific nature of each task. For example, EDF schedulability analyses can consider a large variety of task structures, such as self-suspension times, changing WCET, etc., to reduce the pessimism of the analysis and allow a system designer to utilize task sets which would have otherwise been deemed un-schedulable under a self-suspension unaware analysis. However, these tasks, regardless

of the complexity of the tests and analyses, would still be scheduled by the simple EDF scheduler. A sane EDF scheduler implementation, by itself, would simply handle incoming tasks, re-order them according to their deadlines, select the next task when the currently running task yields (or self-suspends) ad infinitum or until the system is shut down. No scheduling algorithm/implementation is fed a meta-model of the task set to correlate task execution behavior with the behavior expected during the analysis phase.

Therefore, just like the Mad Monk attack and our proposed mitgation strategy discussed in Chapter 5, it is necessary to review self-suspension literature to determine if there is an inherent flaw, from a security perspective, due to the difference in the capabilities of the *scheduler* and the *schedulability analyses/tests*, when the system allows self-suspension. An attacker that controls a self-suspending task, and is able to arbitrarily introduce scheduling points that a scheduler simply responds to, could allow the attacker non-trivial control over system operation. It would be interesting to not only determine whether self-suspension increases the attack surface for real-time systems, but also provide a defense strategy without violating the purpose of explicitly considering self-suspensions, i.e., improving system resource utilization efficiency.

Bibliography

[1] Renesas provides chips for toyota. https://can-newsletter.org/engineering/applications/171114_17-4_renesas-provides-chip-for-toytas-self-driving-cars_renesas, December 2019.

[2] Numicro m2351 series – a trustzone empowered micro-controller series focusing on iot security. https://m2351.nuvoton.com/secure-microcontroller-platform/, October 2019.

[3] Clang 12 documentation, 2020. URL https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[4] Arducopter, 2021. URL https://ardupilot.org/copter/.

[5] Px4, 2021. URL https://px4.io/.

[6] Rover, 2021. URL https://ardupilot.org/index.php/slider/ArdupilotRovers/RubidiumRover.

[7] Turtlebot, 2021. URL https://www.turtlebot.com/.

[8] Fardin Abdi Taghi Abad, Joel Van Der Woude, Yi Lu, Stanley Bak, Marco Caccamo, Lui Sha, Renato Mancuso, and Sibin Mohan. On-chip control flow integrity check for real time embedded systems. In *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 26–31. IEEE, 2013.

[9] N. Almakhdhub, Abraham Clements, S. Bagchi, and M. Payer. µrai: Securing embedded systems with return address integrity. In *NDSS*, 2020.

[10] Sherif Aly. Consolidating AUTOSAR with complex operating systems (AUTOSAR on linux). Technical report, SAE Technical Paper, 2017.

[11] Amazon. Market leading rtos (real time operating system) for embedded systems with internet of things extensions, Nov 2020. URL https://www.freertos.org/.

[12] ARM. Security technology building a secure system using trustzone technology (white paper). *ARM Limited*, 2009.

[13] Ahmad Atamli-Reineh, Ravishankar Borgaonkar, Ranjbar A Balisane, Giuseppe Petracca, and Andrew Martin. Analysis of trusted execution environment usage in samsung KNOX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*. ACM, 2016.

[14] Sanjoy Baruah. Certification-cognizant scheduling of tasks with pessimistic frequency specification. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 31–38. IEEE, 2012.

[15] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo d'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, 2011.

[16] Sanjoy K Baruah. Resource sharing in edf-scheduled systems: A closer look. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 379–387. IEEE, 2006.

[17] Sanjoy K Baruah, Alan Burns, and Robert I Davis. Response-time analysis for mixed criticality systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 34–43. IEEE Computer Society, 2011.

[18] Nicolas Bellec, Simon Rokicki, and Isabelle Puaut. Attack detection through monitoring of timing deviations in embedded real-time systems. In *ECRTS 2020-32nd Euromicro Conference on Real-Time Systems*, pages 1–22, 2020.

[19] Malek Ben Salem. *Towards effective masquerade attack detection.* PhD thesis, Columbia University, 2012.

[20] Jonas Berg, Jens Pommer, Chuan Jin, Fredrik Malmin, and Johan Kristensson. Secure gateway – a concept for an in-vehicle ip network bridging the infotainment and the safety critical domains. 2015.

[21] Richard W Berger, Devin Bayles, Ronald Brown, Scott Doyle, Abbas Kazemzadeh, Ken Knowles, David Moser, John Rodgers, Brian Saari, Dan Stanley, et al. The rad750/sup tm/-a radiation hardened powerpc/sup tm/processor for high performance spaceborne applications. In *2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542)*, volume 5, pages 2263–2272. IEEE, 2001.

[22] Enrico Bini and Giorgio C Buttazzo. Biasing effects in schedulability measures. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, ECRTS 2004.* IEEE.

[23] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[24] B Brandenburg. Schedcat: The schedulability test collection and toolkit, 2013.

[25] Robert Buecs, Pramod Lakshman, Jan Weinstock, Florian Walbroel, R. Leupers, and G. Ascheid. Fully virtual rapid adas prototyping via a joined multi-domain co-simulation ecosystem. In *VEHITS*, 2018.

[26] Alan Burns and Sanjoy Baruah. Timing faults and mixed criticality systems. In *Dependable and Historic Computing*, pages 147–166. Springer, 2011.

[27] Alan Burns and Sanjoy Baruah. Towards a more practical model for mixed criticality systems. In *Workshop on Mixed-Criticality Systems (colocated with RTSS)*, 2013.

[28] Alan Burns and Robert I Davis. Mixed criticality on controller area network. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 125–134. IEEE, 2013.

[29] Alan Burns and Robert I Davis. A survey of research into mixed criticality systems. *ACM Computing Surveys (CSUR)*, 50(6):1–37, 2017.

[30] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*.

[31] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Survey*, 2017.

[32] Giorgio C Buttazzo. Rate monotonic vs. edf: Judgment day. *Real-Time Systems*, 29 (1):5–26, 2005.

[33] Defense Use Case. Analysis of the cyber attack on the ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)*, 388:1–29, 2016.

[34] Donghoon Chang. CAN-FD-Sec: Improving security of CAN-FD protocol. In *Security and Safety Interplay of Intelligent Software Systems: ESORICS 2018 International Workshops, ISSA 2018 and CSITS 2018, Barcelona, Spain, September 6–7, 2018, Revised Selected Papers*, page 77. Springer, 2018.

[35] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno,

et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, volume 4, pages 447–462. San Francisco, 2011.

[36] Chien-Ying Chen, Amiremad Ghassami, Stefan Nagy, Man-Ki Yoon, Sibin Mohan, Negar Kiyavash, Rakesh B Bobba, and Rodolfo Pellizzoni. Schedule-based side-channel attack in fixed-priority real-time systems. Technical report, 2015.

[37] Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–p, 2014.

[38] Kyong-Tak Cho and Kang G Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *USENIX Security Symposium*, pages 911–927, 2016.

[39] Kyong-Tak Cho and Kang G Shin. Viden: Attacker identification on in-vehicle networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1109–1123. ACM, 2017.

[40] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[41] Olivier Cros, Frédéric Fauberteau, L George, and X Li. Mixed-criticality over switched ethernet networks. In *Ada User Journal, Proc of Workshop on Mixed Criticality for Industrial Systems (WMCIS'2014)*, volume 35, pages 138–143, 2014.

[42] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*,

pages 401–416, San Diego, CA, August 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi.

[43] Terry D Day and Sydney G Roberts. A simulation model for vehicle braking systems fitted with ABS. *SAE Transactions*, pages 821–839, 2002.

[44] Dionisio De Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *2009 30th IEEE Real-Time Systems Symposium*, pages 291–300. IEEE, 2009.

[45] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel tsx. In *USENIX Security Symposium*, 2017.

[46] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913, 2015.

[47] Dario Faggioli, Marko Bertogna, and Fabio Checconi. Sporadic server revisited. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 340–345, 2010.

[48] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6):29, 2011.

[49] Mohammad Farsi, Karl Ratcliff, and Manuel Barbosa. An overview of controller area network. *Computing & Control Engineering Journal*, 10(3):113–120, 1999.

[50] Mahsa Foruhandeh, Yanmao Man, Ryan Gerdes, Ming Li, and Thidapat Chantem. SIMPLE: Single-frame based physical layer identification for intrusion detection and

prevention on in-vehicle networks. In *Annual Computer Security Applications Conference*, 2019.

[51] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. AUTOSAR–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, page 5, 2009.

[52] Laurent George, Nicolas Rivierre, and Marco Spuri. *Preemptive and non-preemptive real-time uniprocessor scheduling*. PhD thesis, Inria, 1996.

[53] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. {AutoLock}: Why cache attacks on {ARM} are harder than you think. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1075–1091, 2017.

[54] Bogdan Groza, Stefan Murvay, Anthony Van Herrewege, and Ingrid Verbauwhede. Libra-can: a lightweight broadcast authentication protocol for controller area networks. In *International Conference on Cryptology and Network Security*, pages 185–200. Springer, 2012.

[55] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *DIMVA*, 2016.

[56] Chuancai Gu, Nan Guan, Qingxu Deng, and Wang Yi. Improving ocbp-based scheduling for mixed-criticality sporadic task systems. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 247–256. IEEE, 2013.

[57] Nan Guan and Wang Yi. General and efficient response time analysis for edf scheduling.

In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[58] Yutian Gui, Ali Shuja Siddiqui, and Fareena Saqib. Hardware based root of trust for electronic control units. In *SoutheastCon 2018*, page 1–7. IEEE, 2018.

[59] Zhishan Guo, Kecheng Yang, Sudharsan Vaidhun, Samsil Arefin, Sajal K Das, and Haoyi Xiong. Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 373–383. IEEE, 2018.

[60] Florian Hartwich et al. CAN with flexible data-rate. In *Proc. iCC*, pages 1–9. Citeseer, 2012.

[61] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B Bobba. Contego: An adaptive framework for integrating security tasks in real-time systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[62] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*.

[63] Intel. Control-flow enforcement technology specification, 2020. URL https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[64] Marine Kadar, Gerhard Fohler, Don Kuzhiyelil, and Philipp Gorski. Safety-aware integration of hardware-assisted program tracing in mixed-criticality systems for security

monitoring. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 292–305. IEEE, 2021.

[65] Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. Tzmcfi: Rtos-aware control-flow integrity using trustzone for armv8-m. *International Journal of Parallel Programming*, pages 1–21, 2020.

[66] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[67] Christoph M Kirsch and Ana Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012.

[68] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 39–53. IEEE, 2020.

[69] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2010.

[70] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[71] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. HMAC: Keyed-hashing for message authentication. *RFC Editor*, 1997.

[72] Sreenath Krishnadas. Concept and implementation of AUTOSAR compliant Automotive Ethernet stack on Infineon Aurix Tricore board. Master's thesis, Technische Universität Chemnitz, 2016.

[73] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[74] Jinfeng Li, Liwei Chen, Qizhen Xu, Linan Tian, Gang Shi, Kai Chen, and Dan Meng. Zipper stack: Shadow stacks without shadow. In *European Symposium on Research in Computer Security*, pages 338–358. Springer, 2020.

[75] Giuseppe Lipari and Sanjoy Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, pages 193–200. IEEE, 2000.

[76] Di Liu, Jelena Spasic, Nan Guan, Gang Chen, Songran Liu, Todor Stefanov, and Wang Yi. Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 35–46. IEEE, 2016.

[77] Arm Ltd. Keil rtx5. https://www.arm.com/products/development-tools/embedded-and-software/rtx5-rtos, October 2019.

[78] GG Lucas, D McLean, and P Adcock. Microprocessor controlled fuel injection for automotive diesel engines. Technical report, SAE Technical Paper, 1983.

[79] Vahid Meraji and Hadi Soleimany. Evict+time attack on intel cpus without explicit knowledge of address offsets. *ISC Int. J. Inf. Secur.*, 13:19–27, 2021.

[80] Microsoft. Control flow guard - win32 apps. URL https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[81] Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. TEECheck: Securing Intra-Vehicular Communication Using Trusted Execution. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, 2020.

[82] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey: an efficient MAC algorithm for 32-bit microcontrollers. In *International Conference on Selected Areas in Cryptography*, pages 306–323. Springer, 2014.

[83] Anway Mukherjee, Tanmaya Mishra, Thidapat Chantem, Nathan Fisher, and Ryan Gerdes. Optimized trusted execution for hard real-time applications on cots processors. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, pages 50–60, 2019.

[84] Anway Mukherjee, Tanmaya Mishra, Thidapat Chantem, Nathan Fisher, and Ryan M. Gerdes. Optimized trusted execution for hard real-time applications on cots processors. In *RTNS '19*, 2019.

[85] Sen Nie, Ling Liu, and Yuefeng Du. Free-fall: Hacking tesla from wireless to can bus. *Briefing, Black Hat USA*, 25:1–16, 2017.

[86] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*, pages 479–498, 2013.

[87] Stefan Nürnberger and Christian Rossow. –vatican–vetted, authenticated can bus. In

International Conference on Cryptographic Hardware and Embedded Systems, pages 106–124. Springer, 2016.

[88] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N Asokan. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2017.

[89] Taeju Park and Soontae Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 253–262. IEEE, 2011.

[90] Héctor Pérez, J. Javier Gutiérrez, Salva Peiró, and Alfons Crespo. Distributed architecture for developing mixed-criticality systems in multi-core platforms. *J. Syst. Softw.*, 123:145–159, 2017.

[91] Andreea-Ina Radu and Flavio D Garcia. LeiA: A lightweight authentication protocol for CAN. In *European Symposium on Research in Computer Security*, pages 283–300. Springer, 2016.

[92] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.

[93] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Carlo Contavalli, Amin Vahdat, et al. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 404–417. ACM, 2017.

[94] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza

Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*.

[95] Hovav Shacham et al. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM conference on Computer and Communications Security*, pages 552–561, 2007.

[96] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[97] Vijaya Kumar Sundar and Arvind Easwaran. A practical degradation model for mixed-criticality systems. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 171–180. IEEE, 2019.

[98] Sandra R Thuel and John P Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *RTSS*, pages 22–33, 1994.

[99] José M Urriza, Francisco E Paez, Ricardo Cayssials, Javier D Orozco, and Lucas S Schorb. Low cost slack stealing method for rm/dm. *International Review on Computers and Software*, 5(6):660–667, 2010.

[100] Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. VulCAN: Efficient component authentication and software isolation for automotive control networks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 225–237. ACM, 2017.

[101] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia

Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[102] Anthony Van Herrewege, Dave Singelee, and Ingrid Verbauwhede. CANAuth-a simple, backward compatible broadcast authentication protocol for CAN bus. In *ECRYPT Workshop on Lightweight Cryptography*, volume 2011, 2011.

[103] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE international real-time systems symposium (RTSS 2007)*, pages 239–243. IEEE, 2007.

[104] Kizheppatt Vipin. CANNoC: An open-source noc architecture for ecu consolidation. In *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 940–943. IEEE, 2018.

[105] Robert J Walls, Nicholas F Brown, Thomas Le Baron, Craig A Shue, Hamed Okhravi, and Bryan C Ward. Control-flow integrity for real-time embedded systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[106] Qiyan Wang and Sanjay Sawhney. VeCure: A practical security framework to protect the can bus of vehicles. In *2014 International Conference on the Internet of Things (IOT)*, pages 13–18. IEEE, 2014.

[107] DRM Widevine. Architecture overview, 2017.

[108] Xuhang Ying, Giuseppe Bernieri, Mauro Conti, and Radha Poovendran. TACAN: Transmitter authentication through covert channels in controller area networks. In

Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems, pages 23–34. ACM, 2019.

[109] Joseph Yiu. Armv8-m architecture technical overview. *ARM WHITE PAPER*, 2015.

[110] Mingwei Zhang and R Sekar. Control flow integrity for {COTS} binaries. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 337–352, 2013.

[111] Qinglin Zhao, Zonghua Gu, and Haibo Zeng. Integration of resource synchronization and preemption-thresholds into edf-based mixed-criticality scheduling algorithm. *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 227–236, 2013.

[112] Qinglin Zhao, Zonghua Gu, Min Yao, and Haibo Zeng. Hlc-pcp: A resource synchronization protocol for certifiable mixed criticality scheduling. *IEEE Embedded Systems Letters*, 6:8–11, 2014.

[113] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. Silhouette: Efficient protected shadow stacks for embedded systems. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[114] Tobias Ziermann, Stefan Wildermann, and Jürgen Teich. CAN+: A new backward-compatible controller area network (can) protocol with up to 16x higher data rates. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1088–1093. European Design and Automation Association, 2009.