Securing Safety Critical Automotive Systems

by

Ahmad MK. Nasser

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Information Science)
in the University of Michigan-Dearborn
2019

Doctoral Committee:

Associate Professor Di Ma, Chair
Associate Professor Jinhua Guo
Assistant Professor Bochen Jia
Professor Brahim Medjahed

Ahmad MK. Nasser

ahmadnas@umich.edu

ORCID iD 0000-0001-8318-7082

DEDICATION

This dissertation is dedicated first to my mom and dad: Amal Awada and Mohamad-Kheir Nasser. You have taught me from a young age not only to have big dreams, but also to have the tenacity to pursue them. I also like to dedicate this to my wife, Batoul Abdallah. Your love and support gave me the necessary strength to persevere through the arduous journey of combining a full time career with challenging academic research. Last but not least, I like to dedicate this to my children: Yahya and Dalia. Although, you gave me many reasons to be distracted, I want this achievement to motivate you in your future endeavors to always challenge yourself and aim to leave a mark in whatever you choose to do.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. Di Ma, for her guidance, insights and valuable feedback when reviewing my work. It was her lectures on network security that initially sparked in me the desire to learn more about the intersection of security and safety. Thanks to her advice, I was later able to refine my research skills when looking at problems, not only as an engineer seeking solutions, but also as a researcher looking for answers. Our collaboration made it possible for this work to be impactful. I also would like to express my gratitude to my committee members, Dr. Brahim Medjahed, Dr. Jinhua Guo, and Dr. Bochen Jia, for the time they spent in reviewing my work and the valuable feedback they provided. In addition, this work would not have been possible without the support of numerous colleagues and collaborators who have given me their advice and technical opinions. Special thanks to the Renesas HSM firmware team (Raymon Abdelmassih, Shem Rajiah, Li Li, Cristian Man, and Plamen Stoyanov) for their support. I am also grateful to Renesas for providing me the hardware and software to carry out my experiments as well as the support in pursuing this work, with special thanks to Paul Kanan, Yasuhisa Shimazaki, and Takeo Tomokane. To Eric Winder, Wonder Gumise, and Dr. Matthias Krauledat, I am grateful for our technical debates and discussions which allowed me to refine my ideas for the larger academic audience. Last but not least, I like to thank Dheeraj Sharma, Chris Thibeault, and Kyle Taylor from Elektrobit for their support with the AUTOSAR software stack.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

ABSTRACT

In recent years, several attacks were successfully demonstrated against automotive safety systems. The advancement towards driver assistance, autonomous driving, and rich connectivity make it impossible for automakers to ignore security. However, automotive systems face several unique challenges that make security adoption a rather slow and painful process. Challenges with safety and security co-engineering, the inertia of legacy software, real-time processing, and memory constraints, along with resistance to costly security countermeasures, are all factors that must be considered when proposing security solutions for automotive systems. In this work, we aim to address those challenges by answering the next questions. What is the right safety security co-engineering approach that would be suitable for automotive safety systems? Does AUTOSAR, the most popular automotive software platform, contain security gaps and how can they be addressed? Can an embedded HSM be leveraged as a security monitor to stop common attacks and maintain system safety? When an attack is detected, what is the proper response that harmonizes the security reaction with the safety constraints? And finally, can trust be established in a safety-critical system without violating its strict startup timing requirements? We start with a qualitative analysis of the safety and security co-engineering problem to derive the safety-driven approach to security. We then apply the approach to the AUTOSAR classic platform to uncover security gaps. Using a real automotive hardware environment, we construct security attacks against AUTOSAR and evaluate countermeasures. We then propose an HSM based security monitoring system and apply it against the popular CAN masquerading attack. Finally, we turn to the trust establishment problem in

constrained devices and offer an accelerated secure boot method to improve the availability time by several factors. Overall, the security techniques and countermeasures presented in this work improve the security resilience of safety-critical automotive systems to enable future technologies that require strong security foundations. Our methods and proposed solutions can be adopted by other types of Cyber-Physical Systems that are concerned with securing safety.

CHAPTER I

Introduction

As defined in [60], automotive security is the field of assessing cyber threats against automotive systems and developing security countermeasures for the detection and prevention of the corresponding attacks. Due to the increased connectivity of automotive systems to the internet and smart devices, the threat of cyber attacks is ever more present. Since automotive systems contain an element of control, safety becomes a concern when exposed to security threats, therefore securing these systems is a high priority. Furthermore, securing safety-critical systems in the vehicle requires a holistic approach that takes into account all vehicle access points, external interfaces, and internal networks. Failure to properly secure vehicles will result in a chaotic world where drivers are at the mercy of malicious attackers, who are determined to create safety hazards through cyber attacks. While functional safety in automotive systems is a mature and stable domain, the interplay between safety and security remains a hot topic that requires solving. What makes it a difficult problem to solve is the automotive industry's resistance to radical changes due to massive market forces of cost sensitivity, legacy tools and software, as well as existing infrastructure that cannot be easily adapted. Therefore, security solutions must work within the existing constraints of the automotive industry otherwise face rejection. The primary question we want to answer in this dissertation is how to maintain vehicle safety in

the presence of a malicious attacker who is determined to circumvent typical security countermeasures while working within the aforementioned constraints. To answer this question, we take a two-tiered approach: first, by looking at the system view to formulate a safety aware security engineering method, and second, by zooming in onto the electronic control unit to uncover security gaps and formulate security countermeasures that ensure safe operation even while under attack. To study the safety and security interplay the following road map is followed:

- Formulate a system level approach to engineering secure safety systems
- Evaluate the current state of the art of ECUs in terms of safety and security to find existing gaps
- Close the security gaps with a solution that is compatible with the safety requirements of the target systems
- Build practical security defenses that meet the constraints of real-time systems to satisfy availability and integrity requirements

1.1 Contributions

The contributions of this work are multi-fold. First, we start by studying the Over-the-Air Update process from a safety point of view. We formulate a safety driven security approach that aims to uncover security requirements that protect the safety aspects of the update process in the ECU. We compare the approach against the state of the art OTA industry approach and show comparable findings. The safety-driven approach is further used in our work when defining a secure architecture for protecting safety in deeply embedded ECUs. The second contribution is in the area of AUTOSAR security analysis. We are the first to perform a comprehensive security analysis of AUTOSAR as it relates to safety to identify security gaps. We find that

AUTOSAR classic took a primarily safety-focused approach which led to some design decisions being in conflict with security. We demonstrate a network-based attack that can result in an ECU shutdown. We also show ways in which AUTOSAR safety mechanisms can be leveraged to improve security hardening. The third contribution is in defining a secure ECU architecture that leverages the embedded HSM to provide security monitoring of four critical aspects of the ECU. A context-based control flow integrity monitor safeguards safety-critical program points by monitoring the relevant execution paths. A timing monitor ensures an attacker does not reroute control away to his own malicious code consequently starving safety-critical tasks from execution budget. A firmware integrity monitor ensures flash tampering is detected in real time to prevent persistent code modification. And a CAN peripheral integrity monitor ensures that the message transmit list is not modified after startup to launch a CAN masquerading attack. We show that the approach has minimal impact on legacy code and host CPU run-time. The fourth contribution is in the area of secure boot acceleration. Due to the strict startup requirements of safety systems and the increased demands for memory, authenticating the full software at startup can take prohibitively excessive time leading to compromises in what gets authenticated and when. Our probabilistic approach achieves high confidence in the integrity of the software within a fraction of the traditional secure boot time.

1.2 Organization

The dissertation is organized as follows: Chapter II introduces background topics and related work. In Chapter III, the OTA process is analyzed from a safety point of view to derive the safety-driven approach to security analysis. In Chapter IV, the approach is applied to the AUTOSAR classic software platform to identify security gaps and propose security hardening countermeasures. In Chapter V, the HSM based security monitoring system is introduced to address the security gaps uncovered in

Chapter IV. In Chapter VI, a probabilistic approach to secure boot is presented to enable the accelerated availability of automotive embedded devices. Finally, Chapter VII provides a conclusion and future work.

CHAPTER II

Background and Related Work

2.1 AUTOSAR

AUTOSAR stands for AUTomotive Open System ARchitecture. The AUTOSAR classic version has seen wide success in the automotive ECU market since its initial introduction in 2003 and is expected to expand its dominance as more users continue to adopt it across the automotive industry [1]. One of the greatest strengths of AUTOSAR is its standardized architecture, shown in Figure 5.1, where hardware features are abstracted away from the rest of the system to ease software portability and interoperability. For safety support, AUTOSAR follows an approach of safety element out of context and provides several safety mechanisms to support typical automotive use cases. In terms of security, AUTOSAR provides a cryptography stack which enables ECUs to provision keys as well as execute various security and cryptography services. There is also a secure onboard communication module that provides authentication services to in-vehicle network data. As of version 4.4, additional security modules have been added to support key management, security audit logging, and dynamic rights management for diagnostic access. Beyond that, it is expected that ECU designers build additional security layers to secure their systems, for e.g. locking JTAG ports, implementing secure flash bootloaders, enforcing secure boot on startup, and following security best practices such as the principle of least privilege.

AUTOSAR makes a general assumption that the host CPU running the AUTOSAR stack is trusted and therefore makes no requirement on how security sensitive software is accessed and by whom. In fact, AUTOSAR uses the term "Trusted Applications" to refer to any application partition which is considered safety qualified. As a result, such applications run at the same privilege level as the operating system [11], and can, therefore, change the MPU configuration to disable security protections of memory. It is, therefore, up to the implementer to add the necessary protections to prevent unauthorized access to the security-relevant services of AUTOSAR and to act when an attack is detected. In this work, we study the gaps that currently exist in AUTOSAR and create supplementary countermeasures to increase the security resilience of AUTOSAR based systems.

2.2 Automotive Embedded Systems

Modern vehicles contain upwards of 80 ECUs that control various vehicle functions in a distributed fashion [35]. The right vehicle data architecture is essential to the security posture of these ECUs. For e.g., if the braking control ECU is connected directly to the infotainment head unit, the attacker only has to compromise the infotainment head unit to gain access of the braking controller which can have catastrophic results. In a state of the art vehicle architecture, shown in Figure 2.1, vehicle buses are separated based on the domain function with a central gateway that controls cross-domain message transfer. This provides a level of isolation to prevent a malicious ECU from being able to compromise the entire vehicle. Normally, modules with external connectivity are located behind the gateway ECU. Examples of those are the telematics unit with a cellular connection, the bluetooth and wi-fi radios and V2X module for the car to car communication. On the opposite end of the gateway, various domain networks are separated by vehicle function. The body bus contains ECUs that control comfort such as door control and heating and cooling. The power-

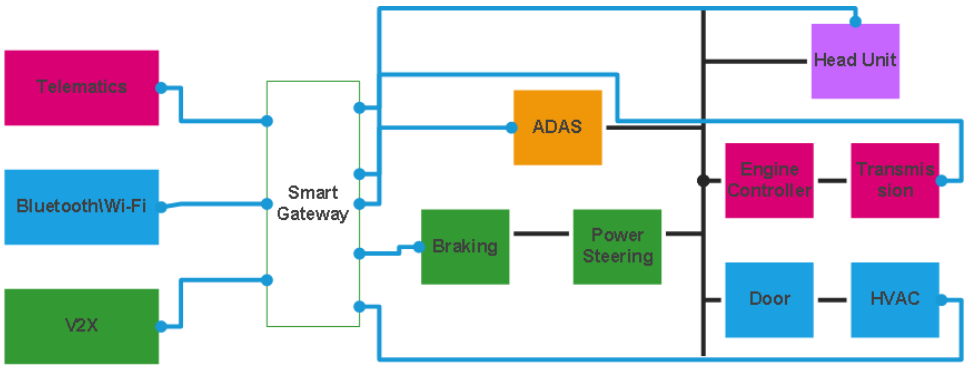

Figure 2.1: Network architecture of a modern vehicle

train bus contains the engine and transmission controllers. The chassis bus contains steering and braking controllers. The ADAS bus contains the advanced driving assistance modules which can leverage cameras and radar. And finally, the media bus contains the head unit which serves various media content to the passengers. The domain networks most popular communication protocol is CAN due to its robustness and low implementation cost. Other bus systems such as LIN, Flexray, MOST and Ethernet are commonly used based on the vehicle manufacturer preference. Data flowing over these buses are mainly used for control, status, diagnostics, and media. It is well understood that a defense-in-depth approach is needed to secure the various data layers of the vehicle architecture as shown in Figure 2.2. First, external interfaces are protected through traditional firewall technology. Next, an intrusion detection or prevention system in the smart gateway ensures anomalous network data is detected or stopped. In addition to that, the gateway enforces separation rules that prevent nodes of one bus from directly sending messages to other networks. Beyond the gateway, individual vehicle bus domains are protected through the AUTOSAR secure onboard communication layer [10], which provides data authentication and freshness protections. Finally, the ECU level security leverages hardware security modules(HSM) in order to establish trust and protect its security assets. Note, most vehicles today only implement a subset of the aforementioned security layers due to the cost and/or impact on legacy systems.

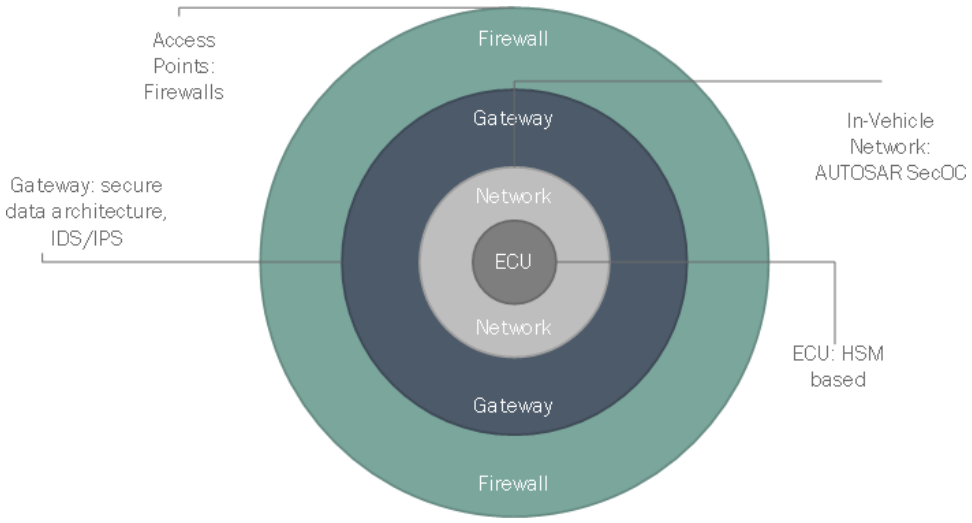

Figure 2.2: Security in-depth concept for advanced vehicle architectures

2.3 Hardware Security Modules

Certain attacks against automotive embedded systems cannot be thwarted through software-only security mechanisms. An embedded HSM plays a vital role in bridging that gap by providing support for security primitives, such as secure boot, secure communication, access control, key management, and cryptographic acceleration. The HSM defined by EVITA [100] and later extended by Bosch [36], was designed to meet the various security use cases deemed relevant for automotive systems. As shown in Figure 2.3, HSMs are embedded in the microcontroller with a dedicated CPU and cryptographic hardware accelerators. Additionally, they support a random number generator along with optional features such as I/O ports. Communication between the host and the HSM is done through an interrupt mechanism and shared RAM buffers. Due to the flexibility of the firmware executed within the HSM environment [29], it is possible to add security extensions such as host CPU monitoring to detect malicious behavior. Depending on the chip, an HSM may be able to monitor several host subsystems such as CAN as well as control the reset of the host CPU.

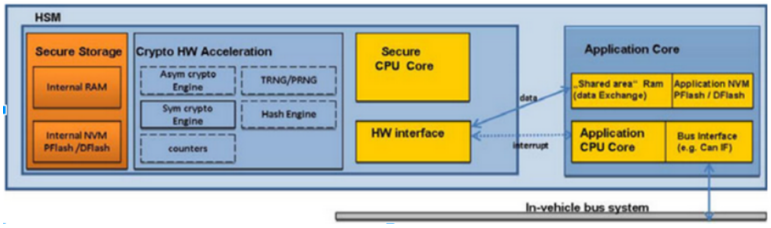

Figure 2.3: Evita Medium HSM architecture

2.4 Combining Safety and Security in Automotive Systems

The problem of combining safety and security engineering processes has been studied in various industries in which cybersecurity threats intersect with the physical control of a process. A survey by Kriaa et al. [65], showed the various standards and approaches proposed across a wide range of cyber-physical systems. Although the automotive domain is relatively a latecomer to this problem, many parallels can be drawn between challenges facing the automotive industry and other industries. Briefly, the problem of security interdependency is one in which the two domains can be in any one of the following states [85], [53]:

- Synergistic: here the safety and security requirements compliment one another. A typical example is the usage of a MAC to protect both authenticity and data integrity which also fulfills the role of a safety CRC. We shall see in Chapter IV, how we leverage this relationship type in AUTOSAR based systems to improve security resilience.

- Conflicting: here a safety and security requirement contradict or compete with one another. We shall see in Chapter IV how this relationship results in safety mechanisms making the system vulnerable to specific attacks.

- Conditional: here safety and security are prerequisites for one another. We shall see in Chapter V how we build security systems that guarantee this relationship type is upheld when building security countermeasures that respect safety

constraints.

For automotive systems, Glas et al. [53], presented various cases in which these relationships can be observed. In terms of conflicting requirements, they gave the example of a safety-related cyclic RAM test that requires stalling the HSM core to prevent interference. As a result, the safety requirement violates a security principle that a security module(e.g. HSM) is disabled during normal operation. Such scenarios are important to detect early on in the design of the vehicle to avoid introducing bugs, or security vulnerabilities. The work in [101] took a practical approach to assessing the resilience of safe automotive micro-controllers in handling power glitching attacks. They showed that the dual lock core step safety mechanism which is meant to detect faults within the CPU, could be leveraged to detect maliciously induced CPU output mismatches. However, the safety mechanism on its own was not always able to detect all the power glitches which lead to skipping critical security code sections that unlocked the JTAG interface. The work highlighted the potential for synergistic relations between safety and security mechanisms albeit with some effort to account for malicious fault sources. In the area of AUTOSAR based safety and security overlap, the work of [34], showed how the EB tresos [2] AUTOSAR OS can be used to harden the ECU against attacks. Using mechanisms such as stack overflow protection, and MPU based read/write and execution prevention, the security posture of such systems can be significantly improved. The work in [56] raised the issue of conditional safety and security when introducing TPMs in a safety-related Cyber-Physical System. The study showed various aspects of potential conflicts if the safety element is dependent on the TPM. As a result, a set of requirements was proposed to ensure TPM technology can be used safely. The study of TPM usage in a safety context is useful to explain the practical results of introducing security elements which can interfere with the safety concept of the system and thus require adequate analysis and handling.

In terms of the interplay between safety and security engineering processes in Cyber-Physical Systems, there are generally two schools of thought. The unification approaches aim to produce a single engineering process that considers both safety hazards and security threats simultaneously as in [95] and [31]. While this reduces the number of engineering resources, it is challenging to find experts who can tackle both areas with the same required rigour. The integration approaches on the other hand, propose separating the two processes while cross-referencing the resulting requirements to identify and resolve conflicts as in [59]. The automotive industry has also attempted to address the engineering processes interdependence through various standardization efforts. The automotive functional safety standard, ISO 26262 [17], requires that hazards to functional safety be adequately addressed to reduce their risk to acceptable levels. In terms of security, the standard leaves the handling of threats that cause safety hazards open for future standards to handle. This is where SAE stepped in through SAE J3061 [43], to define a cybersecurity framework that is analogous in process steps to the safety standard. In terms of the combination of safety and security, the standard proposes three ways of approaching that:

- Apply the process as a standalone activity with communication points to the safety standard
- Apply process steps in conjunction with the safety standard
- Apply the process as a hybrid, which leads to some activities being done in conjunction and others done separately

Due to the vast difference of expertise between the domains of safety and security, there is a near consensus in the automotive industry that the ideal approach pursues a parallel path between the two processes with connection points that ensure consistency between the derived safety and security requirements. SAE J3061 extends the Hazard Analysis and Risk Management (HARA) from functional safety, into security

Threat Analysis and Risk Assessment (TARA), to identify and prioritize cybersecurity risks. Over the years, several TARA methods have been proposed for automotive systems such as HEAVENS [3], SAHARA [71], and EVITA [87]. They follow a general scheme of applying a threat model against system components or functional use cases to derive relevant threats. Then they decompose each threat based on a specific risk model to allow prioritization and subsequent mitigation. Threats that result in severe impact to safety while requiring low resources, and know-how are prioritized for mitigation. Such threats can result in changes to the safety architecture to mitigate the corresponding security risks. HEAVENS, which stands for Healing Vulnerabilities to Enhance Software Security and Safety, uses the STRIDE threat model [91], against functional use cases to derive threats. Then decomposes threat levels in terms of required expertise, knowledge about the target, window of opportunity to carry out the attack, the type of equipment needed to launch the attack, and the impact severity. SAHARA follows a similar approach but for threat assessment, it considers three factors: required resource to materialize the threat, required know how, and threat criticality. EVITA [87], emphasizes the use of attack trees to derive attacks and performs the risk analysis based on an ASIL extended risk model. The work in [38], presented an approach to extend the ISO 26262 safety standard [17], to consider security threats and their potential hazardous classification. They apply their approach to the use case of the adaptive cruise control function. This resulted in deriving a set of functional and technical security requirements which aim to reduce the risk of security threats to an acceptable level. The approach highlighted again the many parallels between safety and security engineering processes. Alternatively, the authors in [26] proposed an iterative approach to safety and security in which one process provides feedback to the next. Their approach requires a safety pattern engineering step to be performed first to arrive at the close to complete architecture. Next comes the security pattern engineering which can result in impacts on the safety

architecture. This is followed by a safety and security co-engineering loop to resolve conflicts between the two. The iterative nature of that approach is closely related to our safety-driven approach to security(SDAS) which we introduce in Chapter III. The gap in the available approaches listed above is that no clear direction is given on how to link the safety and security processes practically. While the need for harmonizing the security and safety requirements is well-understood, how one would go about doing so is left open. In Chapter III, we shall see how our proposed SDAS approach tackles this problem by providing a practical method for linking the safety engineering analysis to the security analysis in a way that produces harmonized requirements between the two domains. Note, at the time of writing this thesis, the ISO/SAE 21434 Road Vehicles Cybersecurity engineering standard was not yet published but is expected to provide answers that relate to this topic.

2.5 Attacks on Safety

Since one of the primary goals of our work is to improve the security resilience of safety-critical ECUs, it is useful to survey the types of attacks that such systems are subjected to. The past years have seen a sharp increase in academic publications about successful attacks on automotive systems with the intent to create a safety hazard. Checkoway and Koscher [39] performed a comprehensive analysis of automotive attack surfaces in a groundbreaking study that demonstrated the reality behind cyber attacks on vehicle systems. They showed different ways by which an attacker can get access to the vehicle CAN bus where the critical control commands exist, in order to take control of the vehicle. The approach of their analysis was based on extracting the ECU firmware and then reverse engineering the code and data using disassembly, and debug tools. By disassembling the code using a tool like IDA Pro they could map the control flow and identify vulnerabilities. This approach led to uncovering several vulnerabilities, such as a buffer overflow in the media player which allowed a mali-

cious WMA audio file to run malicious code. The latter could then interfere with the vehicle based on the vehicle architecture and what other ECUs are connected to the media player. This and other vulnerabilities demonstrated how security attacks could be launched to impact vehicle safety. Then came Miller and Valasek [73], [74] who demonstrated a remote attack on a Jeep vehicle that led to the loss of brake function at low speed. This later resulted in a recall of Jeep vehicles and became the most publicized successful attack on a vehicle. They used a software update vulnerability in the head unit in order to modify the firmware of the vehicle processor interface. The latter could then interfere with the CAN bus via the park assist module by sending a diagnostic command to bleed the brakes causing the vehicle to lose braking ability. The study presented many insights into tools and methods by which vehicle security can be analyzed for the purpose of launching a successful remote attack. Following that, several research teams demonstrated successful attacks such as the ones against the Nissan Leaf [63], GM Corvette [54] and Tesla [66], [98]. The Tencent Labs attacks followed a similar pattern of finding a vulnerability in the autopilot electronic unit in order to launch CAN spoofing attack on critical safety components like steering. In response to such attacks, several security best practices have been published [25] and individual OEM's have taken concrete steps to harden the security of their vehicles. Still, the road to fully securing vehicles seems long and arduous. In the following chapters, we demonstrate how security resilience of automotive safety systems can be improved to handle those types of attacks while keeping in line with the unique constraints and challenges of automotive systems.

CHAPTER III

Safety Driven Security Analysis: A System View

3.1 Introduction

The first phase of our research consisted of surveying the available safety-security engineering approaches. As discussed in section 2.4, it was apparent that engineering secure safety-critical automotive systems still faced several questions that required answering. Since the primary goal of our target systems is to ensure safety, the approach applied had to be heavily biased towards that aim. To do so, the following questions shall be answered:

- How can the safety analysis be linked to the security analysis in a way that guarantees that security measures are addressing maliciously triggered hazards?
- In case attack prevention is possible, how to ensure that the resulting security countermeasures are still compatible with the safety architecture?
- When only attack detection is possible, what is the proper reaction that the system shall take to ensure safe operation while under attack?

To answer these questions and formulate a safety and security co-analysis approach, we chose to study the use case of over-the-air updates (OTA). By studying the OTA safety and security interplay, we can formulate then validate our approach through

comparison to the state of the art in OTA security. The outcome of our analysis shall generate security requirements that ensure the process of updating software in the vehicle does not result in an unsafe vehicle state at the ECU level [79]. Following that step, the approach can then be applied to other aspects of our research into ECU level security as will be shown in the following chapters.

3.2 Background

Securing Cyber-Physical Systems (CPS) depends largely on keeping such systems up to date with the latest security patches. A connected vehicle which contains a number of independent and interconnected modules is an example CPS where securely updating the software in any module is a concern for the overall integrity and availability of safety functions. A connected software update process is defined as a process in which the software update is delivered to the vehicle wirelessly or through a connected device to be later flashed into a target ECU over the vehicle network. Reduction of recall costs, the migration towards ADAS and autonomous driving, the desire to download security patches, as well as the possibility of upgrading vehicle features wirelessly, all make the connected software update a mandatory building block to support the future of vehicle development. Until recently, flashing an automotive ECU after it left the manufacturing plant was only possible through the use of a diagnostic tester tool which physically connected through the OBD II port [57], or insertion of physical media into the vehicle's media-related components. Security vulnerabilities with such tools are well-documented in[39]. They showed various attack scenarios such as infecting the diagnostic tool to perform unauthorized diagnostic services or installing malware on an audio CD infect the head unit and gain access to the internal vehicle bus system. Adding connectivity to a vehicle makes the attack surfaces for the software update process much larger. Malware may be delivered to the vehicle when a vulnerability exists in the OTA update/remote diagnosis process,

flaws in embedded web browsers, malicious aftermarket equipment, or through removable media ports and others [103]. With a compromised software update process an attacker can reprogram ECUs with malicious code that can compromise vehicle safety. The famous Jeep UConnect attack by Miller and Valasek [74], relied on a software update vulnerability in the head unit in order to craft a series of attack steps that lead to the modification of the vehicle processor interface firmware. The latter could then interfere with the vehicle CAN bus by sending a diagnostic command to bleed the brakes causing the vehicle to lose braking ability at low speeds. This and other similar proven attacks demonstrate the need for the download process especially at the ECU level where malicious downloads can be catastrophic.

3.2.1 OTA Architecture

The connected software update allows the remote delivery of software updates to an ECU in a vehicle. Whether the update is delivered over the air, for example through a telematics unit or through a Bluetooth enabled smart device, direct authentication with the originating source is needed to ensure the integrity and authenticity of the software update. In this chapter when we discuss over the air update, we are referencing the process by which software updates are delivered to a vehicle wirelessly first by downloading to a primary ECU like the vehicle gateway, and second by flashing the target ECU. The architecture, shown in Figure 3.1, is a simplified diagram of the vehicle data architecture with the following associated attack surfaces and threat sources:

OBD-II Port:

- Download tool connected to the OBD-II connector
- Aftermarket equipment connected to OBD-II port (Insurance Dongle)

Telematics/Headunit:

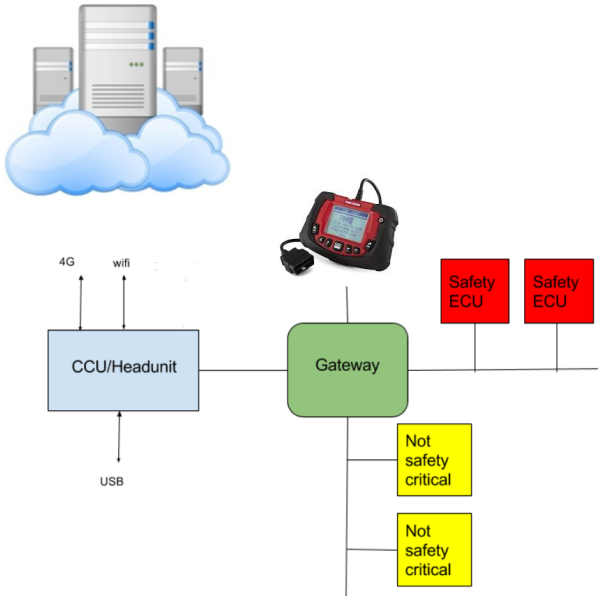

Figure 3.1: Architecture of OTA system considered for the security analysis

- Cellular Modem Connection
- Wifi Hotspot (and related software)
- Bluetooth connection to mobile device
- Media Ports (USB, SD Card, CD/DVD)

Internal Vehicle network:

- CAN bus network
- Compromised ECU, either gateway or ECU on safety bus.

In the following sections, we will see how this architecture results in a set of threats that have safety impacts and how they can be mitigated by applying the safety-driven security approach.

3.3 Related Work

In Section 2.4, we surveyed the various techniques for combining safety and security analysis. Here, we shall look at existing approaches for securing software update

processes to provide a reference for our comparison. Several papers have already presented concepts for a secure software update process as well as the possible threats that such systems face. The EVITA project [87], presented ten attack trees based on eighteen security use cases. One of the security use cases is the attack on the flashing capability of the vehicle. Based on the derived attack tree, security requirements were developed to protect vehicle assets against flash programming threats. The project derived the following security requirements for firmware updates relative to the OEM, the Download Tool (DT), the Communication Control Unit (CCU) and the ECU:

- Firmware confidentiality (CSR.1): Firmware must be kept confidential when transferred from OEM to DT and from DT to the target ECU
- Key confidentiality (CSR.2): Exchanges transferring or using non-public keys must preserve key secrecy along the whole flashing process
- Internal Authenticity (ASR.1): Whenever an exchange between CCU and ECU happens, the correspondence between claimed and real authors must be authenticated
- External Authenticity (ASR.2): Whenever data is exchanged between DT utility and OEM server or between DT and in-car components, the correspondence between claimed and real authors must be authenticated.

Note, the above requirements do not make a distinction between safety-relevant security requirements and those which are purely security relevant. Nilsson et al. [82], studied the different security threats that face software updates and presented a list of threats based on the different elements involved in the flash update process, which are: the portal, the communication link between the portal and the vehicle, and the vehicle itself. In the portal, they pointed to masquerading attacks in which an attacker takes-over or creates a fake portal to distribute malicious software packages. In

the communication link, they pointed to the risk of the wireless link being vulnerable to traffic manipulation through packet injection and replay attacks. As for the vehicle level threats, they pointed to the case where an attacker can impersonate a vehicle and connect to the portal to uncover security vulnerabilities or attack vectors. These vectors can then result in further intrusion attacks. The authors derived security requirements to address those risks in a methodical fashion:

- Preventing impersonation of the portal by using certificate based identity verification of the portal and preloading the portals public key in the vehicle
- Usage of an Intrusion Detection System in the portal to detect an intrusion and take the proper action
- Secure end-to-end communication over the wireless link by encrypting all the data with a freshness counter to prevent packet injection or playback
- Preventing impersonation of a vehicle by establishing the vehicle identity through a certificate
- Usage of a firewall and an IDS in the vehicle to alert against an intrusion and log the intrusion to prevent future breaches

In the above requirements, safety is again assumed as a by-product of securing the overall system. Note, in our work, while we are interested in the overall security, the focus of our work is on securing safety at the ECU level. UPTANE [61], is an open source project that proposes a full OTA solution specifically designed for automotive systems. The project enlisted security experts from academia as well as experts from the automotive industry to come up with a state of the art open source solution to OTA. The project presents the following design goals to deal with security attacks:

- Leverage additional storage to recover from endless data attacks. This allows the new software image to be flashed without overwriting the original image

which can serve as a backup.

- Broadcast metadata to prevent mixed-bundles attacks where a compromised primary ECU can send incompatible software images to secondary ECUs
- Utilize a vehicle version manifest to detect partial bundle installation attacks which can leave ECUs partially programmed
- Use a time server to limit freeze attacks to prevent an ECU from completing its download or preventing it from receiving any updates

The above design goals guide the full implementation of UPTANE which defines in greater detail the roles of primaries, secondaries, and the content of metadata. Due to the maturity and completeness of UPTANE, we shall use it in this chapter as a reference for validating the output of our security analysis using the safety-driven approach to security.

3.4 A Safety Driven Approach to Security

The safety-driven approach to security(SDAS), assumes that the safety analysis is performed initially to define a safe system architecture along with the risk of safety hazards that shall be mitigated. Therefore, SDAS starts from the point where the safety analysis was finished and then re-iterates until both the safety and security requirements are in no further conflict as shown in Figure 3.2. In each iteration, changes applied to the architecture are evaluated from both the safety and security points of view. The approach builds upon the premise that hazards outside the physics of the system cannot be added through cyber attacks. Instead, the latter is only able to manifest hazardous events from the set of possible hazards that a specific ECU can experience. Consider for e.g. an electronic power steering ECU which is required to mitigate the hazard of an over-steering event that can result from a failure in the

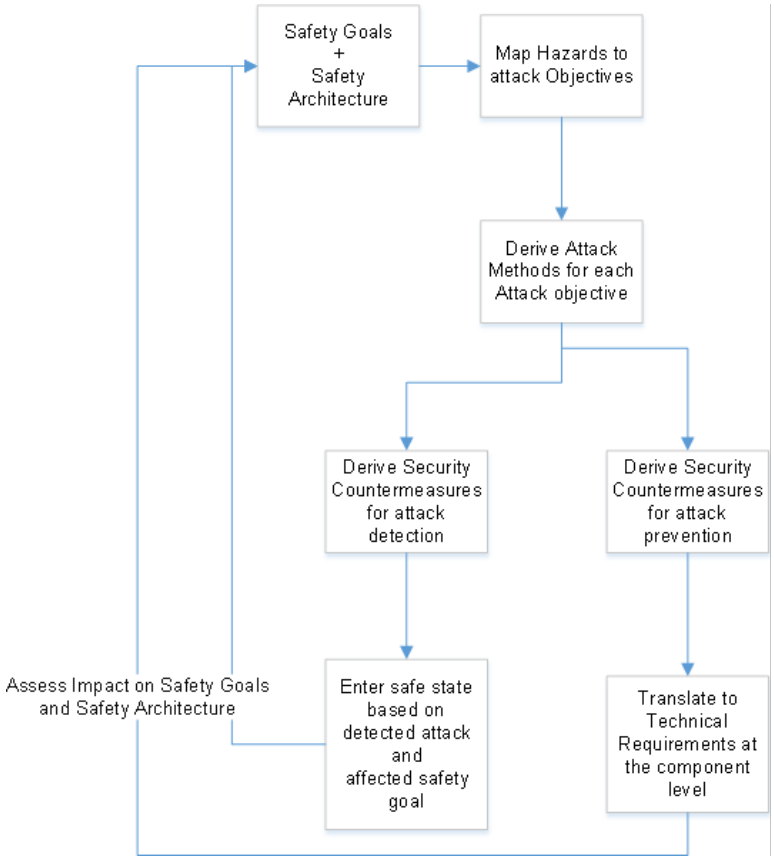

Figure 3.2: Safety Driven Security Approach process steps

steering angle sensor. An attacker can induce the hazard of over-steering by spoofing the sensor data, but he cannot create a totally new hazard in the target ECU outside its own physical limits. Incidentally, he can create hazards for other ECUs through a masquerading attack, but that is handled separately within the hazard analysis of the respective ECUs. Furthermore, by focusing on the safety goals as a starting point we limit the search space of security threats to ones that are guaranteed to result in a safety impact. Since our aim is to secure safety-critical systems, we argue that biasing the security analysis in this manner is acceptable.

Based on the target application use case, we map hazards to attack objectives using the attacker model. Given the attack objectives, there are several techniques to produce attack methods, such as attack trees, as is done in the EVITA project [87], or by looking at documented attack methods as in [92]. Alternatively, applying a

Hazard	Functional Unit
Software is corrupted during the download process without detection by the ECU. ECU then executes corrupted code which can result in undefined and potentially unsafe behavior	CCU,Gateway,Target ECU
Download of incompatible binary files for e.g. a calibration file with unsafe parameter settings	Gateway,Target ECU
A communication failure prevents the download from completing leaving a safety critical ECU un-programmed	Gateway,Target ECU
A power failure during programming leaves a safety critical ECU partially programmed or un-programmed	Gateway,Target ECU
A permanent communication failure in the system prevents any updates from reaching the vehicle including safety critical patches, e.g. recall related updates	CCU

Table 3.1: Simplified Hazard Analysis for OTA

threat model such as STRIDE[91], can also be useful in enumerating threats that can lead to a safety hazard. Although we are focusing here on threats that lead to hazards, this does not mean that non-safety related threats are to be ignored. For such threats, traditional security engineering approaches can be applied in parallel to protect properties such as driver privacy. Once, we have a set of attacks that can directly violate our safety goals, we can now evaluate whether each attack is preventable, or only detectable. For preventable attacks, the next step is to define the technical security requirement at the component level to realize the security countermeasure. If only detection is possible, we need to define the detection method as well as how the system shall react when the attack is detected. Since each attack is mapped to a specific safety goal, we can reuse the fail safe-states defined in the safety case [17], to determine what the proper reaction shall be in the face of such an attack. At this point a re-evaluation of the safety architecture and relevant safety goals is performed

to ensure that the resulting system with the added functional security requirements does not introduce conflicts with the safety assumptions of the system. The process is repeated with several iterations to allow for refinements to the functional safety and security requirements until all conflicts are resolved. One example of such a conflict would be a functional security requirement that aims to immediately reset the system when an attack is detected. This in turn violates a safety goal for a graceful shut-down to prevent sudden loss of system function. Since our analysis is biased towards safety, we always give priority to safety requirements when a conflict is detected. In the remaining sections, SDAS is applied to the connected software update use case to demonstrate its effectiveness.

3.4.1 Safety Goals

As stated above, a prerequisite to analyzing the security threats which have safety implications on the software update process is to define the safety goals first. Typically these are produced during the safety assessment. Since we do not have a real system to analyze, we came up with safety goals based on a simplified hazard and risk analysis of the OTA system described in Figure 3.1. The identified hazards are shown in Table 3.1. Note, we did not assess risk levels of safety goals in this theoretical use case because it requires a complete system to be defined first. The list of hazards we identified lead us to the following consolidated safety goals for the OTA system:

- Data corruption in software downloaded to the ECU shall not go undetected
- Incompatible software images shall not be downloaded to an ECU without detection
- Safety critical ECUs shall not be permanently left in an un-programmed or partially programmed state as a result of an OTA event.

Attacker	Objective	Impact
Organized Crime	Download Malware to control an individual vehicle of victim	Injury or Loss of Life to Individual Car
Terrorist	Download Malware to control fleets of vehicles to cause mass chaos	Injury or Loss of Life to civilian or government vehicles
Cyber Activists	Disrupt OTA process to prevent safety critical patches or vulnerability fixes from being installed	Loss of customer confidence in OTA, potential for injury or loss of life if manual update not performed
Cyber Criminals	Re-program ECUs remotely with unsigned software, to force the ECU to stay in bootloader mode(unprogrammed state) consequently, without safety critical functionality	Loss of consumer confidence in ADAS systems, and increased risk of injuries or loss of life

Table 3.2: Safety Driven Attacker Model

- It shall always be possible to download an update to fix unsafe software or roll back to a safe version

3.4.2 Attacker Model

In our safety driven security approach, the attacker objectives are to trigger hazardous events that violate the defined safety goals. This is presented in Table 3.2. Given our attacker objectives, we can derive three safety driven attack objectives associated with the connected software update as shown in Table 3.3.

3.4.3 Security Requirements

Depending on the security architecture, certain attacks may not be prevented but at least they can be detected and action is taken. A security analysis shall evaluate each attack in terms of feasibility of both, prevention and detection in order to assign the proper recovery action. Based on the attacks presented in Table 3.3, we propose

Attack Description	Safety Goal
Download malware to take control of an ECU without detection	1
Download a validly signed binary images with incompatible software	2
Disrupt the update process indefinitely to prevent an update from reaching completion. Here the attacker may not be able to forge an authentic software image, yet he is still able to interfere with the software update process in a way to prevent critical software updates from being delivered or flashed to the target ECU.	3
Manipulate the update process to result in an unprogrammed ECU. In the case where the attacker cannot forge the signature of the software, he may try to disrupt the download so as to result in an un-programmed ECU.	4

Table 3.3: Attacks with safety goal violations

the following functional security requirements:

- The software download shall only be possible with authenticated data and from an authorized source
- The authenticity of the binaries shall include the compatibility information among binaries to prevent flashing incompatible software binaries
- The OTA architecture shall permit multiple download channels to prevent safety critical updates from being blocked indefinitely through a single compromised channel
- Failure to update safety critical ECUs shall not result in an un-programmed ECU, i.e. the system shall retain a backup copy of the software to be able to restore such ECUs to their previous functioning state

3.4.4 Impact on Safety Architecture

Following the generation of attack prevention and detection countermeasures, SDAS requires evaluating the impact on the original safety architecture to resolve potential conflicts. The derived requirements in the previous section focused completely on prevention. To support these requirements, the ECU requires the ability to perform data authentication and access control functions. This results in the addition of security algorithms as well as hardware support, such as a TPM, to store keys securely. Hoeller et al. [56], showed that TPMs used in safety-critical systems require special handling. The areas of concern are the impact on real-time performance and the potential for faults in the TPM hardware which can result in safety violations. This highlights the need for further refinement of the safety and security architecture to ensure the impact on both is understood and action is taken. In terms of attack detection, when a software image is maliciously erased and the updated image is detected as invalidly signed, the safe state is to revert back to the backup software through the redundant storage. This ensures that our safety goal of not leaving a safety critical ECU in a permanently un-programmed state is upheld.

3.5 Comparison to UPTANE

The derived security requirements at the ECU level map very well to the security requirements stated in UPTANE [61], as shown in Table 3.4. In the following chapters, we shall apply SDAS to a wider use case to validate its effectiveness experimentally.

3.6 Conclusion

In this chapter, we presented a safety-driven approach to security analysis to bridge the gap between the two interdependent domains of functional safety and cybersecurity. We applied the approach to a popular OTA update process use case

UPTANE	SDAS
Prevent endless data attacks through backup storage	Failure to update safety critical ECUs shall not result in an unprogrammed ECU, i.e. the system shall retain a backup copy of the software to be able to restore such ECUs to their previous functioning state
Prevent mixed bundle attacks through the broadcast of meta data	The authenticity of the binaries shall include the compatibility information among binaries to prevent flashing incompatible software binaries
Detect partial bundle installation attacks by using a vehicle version manifest	Failure to update safety critical ECUs shall not result in an unprogrammed ECU, i.e. the system shall retain a backup copy of the software to be able to restore such ECUs to their previous functioning state
Limit freeze attacks by using a timeserver	The OTA architecture shall permit multiple download channels to prevent safety critical updates from being blocked indefinitely through a single compromised channel

Table 3.4: Attacks with safety goal violations

to fine-tune and validate the approach. As a result, we produced a list of security requirements that were shown to be aligned with the state of the art in OTA systems security. By focusing on the safety goals first, we were able to leverage the safety analysis in deriving safety hazards that can be induced maliciously. This led to defining security requirements that are necessary to counteract such threats. Also, by differentiating preventable from strictly detectable attacks, we were able to reference back the safe states of the system to ensure recovery action is compatible with the corresponding safety goals. In the next chapter, we apply SDAS to the AUTOSAR software architecture to uncover safety-relevant security vulnerabilities and

recommend ways to improve the AUTOSAR security resilience.

CHAPTER IV

AUTOSAR Safety and Security: Synergies and Conflicts

4.1 Introduction

In the previous chapter, we explored how taking an iterative approach to safety and security by linking safety goals to security threats can lead us to derive safety-relevant security countermeasures in a harmonized fashion. We saw that a safety-driven approach to security(SDAS), can indeed generate security requirements that safeguard safety as a system property. The approach also takes care of exploring areas of incompatibilities between the safety and security countermeasures which required iterative refinement to arrive at a system that satisfies both objectives. In this chapter, we apply the approach to AUTOSAR as the ECU level software architecture with the hope to uncover security vulnerabilities and then propose security countermeasures that improve the security posture of AUTOSAR based systems. The latter being the most popular software architecture platform for automotive systems [1], is a natural pick when studying the state of the art of automotive systems and ways in which safety and security may be in conflict or for that matter in harmony. Just as in chapter III, we start with the safety analysis since the systems primary function is to meet safety goals. Although, we did not have access to the AUTOSAR safety

analysis documentation, by examining the AUTOSAR defined safety mechanisms we could synthesize the assumed safety architecture behind AUTOSAR safety systems and evaluate how it would fair under the threat of attack. In parallel, as we analyze the security requirements of AUTOSAR, we go back to the safety architecture to determine incompatibilities and synergies. As will be shown in this chapter, there were instances in which safety mechanisms became attack vectors, and in others, they were useful as security countermeasures. When it came to attack mitigation, harmonizing the security response with the safety requirements became a must to ensure a fail-safe attack response. Next, we enumerated safety mechanisms that are synergistic with security and provide some constraints on how they can be used for ECU hardening. We then shift from analytical to experimental validation by demonstrating two attacks against a safety rated micro-controller. Finally, we demonstrate how specific safety mechanisms that are synergistic with safety can be used for attack mitigation, using the same hardware target.

4.2 Attack Model

Deeply embedded systems are assumed to be located behind several defense lines: firewall, security gateway, and a secure communication bus. Assuming the vehicle has implemented a properly layered secure architecture, launching successful attacks on such systems requires compromising several security layers upstream. Previous works by [73], [98], have shown that at some point these defenses can be broken and an attacker may be able to launch a successful attack against the vehicle control system. Let us consider the three classes of attacks that are relevant to both deeply embedded ECUs and traditional computer systems [48]:

- Malware and exploitable software vulnerabilities which aim to take control of the system

- Physical access type attacks
- Network based attacks

Note in the first class, there is a significant difference in the attack surface between traditional computers and deeply embedded systems [39]. In the former, exposure to software exploits is more common due to the rich space of applications that can be loaded and executed on highly configurable operating systems like Linux. In contrast, deeply embedded systems execute a limited pre-defined set of applications from flash memory. Creating persistent malware in flash requires the tampering of the flash bootloader or an exploit that can bypass security checks to use the bootloader routines directly. Alternatively, loading temporary malware requires injecting code in data memory (such as the stack) through a buffer overflow exploit due to a software bug as demonstrated in [50]. When network access is considered together with software-based exploits, deeply embedded systems suddenly become exposed to similar types of attacks as traditional computer systems. While security experts may argue that many protections are already being designed to harden automotive systems, lack of maturity of cybersecurity principles within automotive ECUs gives us the intuition that software vulnerabilities and back doors will persist for several years. For the rest of this chapter, we assume our attackers have indirect network access to the ECU through a compromised link within the vehicle network architecture. The in-vehicle network is assumed to support AUTOSAR SecOC [10] to ensure CAN bus authentication and freshness. Furthermore, there may exist an exploit or backdoor in the ECU which allows loading software on the target. The attacker's objective is to disrupt safety critical systems to create safety hazards.

Module	Mechanism	Max Error Response
E2E	CRC:data integrity	Disable consuming function
E2E	Sequence counter: message order	Disable consuming function
E2E	Alive counter: data freshness	Disable consuming function
E2E	Data Id: detect I-PDU sent on wrong message	Disable consuming function
E2E	Timeout monitoring: detect message loss	Disable consuming function
WdgM	Monitor aliveness of supervised entities	Reset
WdgM	Detect timeout of supervised entity	Reset
WdgM	Monitor control flow of supervised entity	Reset
OSTiming	Monitor task/ISR execution budget	OS Shutdown
OSTiming	Monitor task/ISR inter-arrival time	OS Shutdown
OSTiming	Locking time protection	OS Shutdown
OSMemory	Stackoverflow detection	Reset
OSMemory	Detect execution from data section	Reset
OSMemory	Detect access to restricted memory	Reset
OSHardware	Prevent untrusted apps from accessing privileged HW	Reset
CoreTest	Test health of core MCU components	Reset
RAMTest	Test health of RAM cells	Reset
FlashTest	Test health of Non-volatile memory	Reset

Table 4.1: Survey of AUTOSAR safety mechanisms

4.3 Survey of AUTOSAR Safety Features

Before we can evaluate attacks on safety in AUTOSAR based systems, we start by surveying the safety mechanisms and the corresponding maximum safe state action, as shown in Table 4.1. This will enable us to extrapolate the safety goals in order to apply SDAS and derive security attacks with a safety impact.

Safety Mechanism	Safety Hazard
Cyclic Redundancy Check(CRC): detects data corruption by comparing CRC value appended to message against the calculated CRC	Consume corrupted data
Sequence counter: detects out of sequence messages	Consume out of order data when order is important
Alive counter: detects unchanging(stale) data	Operate on old data that no longer reflects the vehicle state
A unique ID for Interaction Layer Protocol Data Unit (IPDU) group: detects a fault of sending IPDU on unintended message	Misinterpret data from one message as belonging to another message
Timeout monitoring: detects communication loss with the sender	Operate on old or no data due to loss of communication

Table 4.2: Applying SDAS to E2E Library

4.3.1 End to End Library

The first AUTOSAR safety module that we examine is the End to End (E2E) library which defines several protection profiles for data transmitted over a communication channel both internally and externally [13]. The goal of this module is to prevent safety-critical functions from operating on faulty or missing data. To illustrate how SDAS can be applied, we list each safety mechanism and the corresponding hazard it aims to prevent in Table 4.2. Given the extrapolated hazards we apply the STRIDE model to derive several attacks classes that can materialize those hazards. The first four hazards can be caused by data spoofing and tampering attacks. Without proper data authentication, an attacker can tamper with data, spoof an invalid sequence or alive counter as well as the IPDU ID. To cause a communication loss, the attacker can launch a DoS attack to disrupt the transmission of network messages. To mitigate the risk of the first four attacks, a message authentication code(MAC) can be appended to the message to protect data, counters, and identifiers. Since this countermeasure prevents the attack, no fail-safe action is needed. However, the impact of introducing a MAC has to be evaluated against the original safety archi-

tecture. One prominent issue is the MAC calculation time impact. Unlike a CRC, the time of verifying a MAC is not negligible, therefore, the overall overhead has to be factored into the design of network messages to ensure control functions are able to receive their data in a timely manner. Addressing the DoS attack requires the addition of an intrusion detection system to detect anomalous CAN messages, which translates into a change in the overall system architecture. DoS attack prevention may not always be possible, therefore, the system shall at least aim to detect the attack and take action. If an ECU experiences message loss due to a DoS attack, the same fail-safe action can be taken as with normal message timeout faults by shutting down the relevant safety functions. The introduction of the IDS requires an additional iteration to evaluate the impact of this change on the overall safety assumptions of the original system. The safety architecture must now account for possible data loss not only due to a real DoS attack but also due to the IDS falsely flagging valid CAN messages as anomalous. The risk of false positives has to be assessed from a safety point of view as it correlates to a system fault. This process continues until all the safety and security requirements are satisfied.

4.3.2 AUTOSAR OS Protections

AUTOSAR OS [11] defines the properties and interfaces of a real time operating system. The standard also defines protection mechanisms that are essential for building safety critical applications. Those protections fall under the following categories:

- Memory Protection: to provide freedom of interference between OS applications and tasks of mixed criticality
- Timing Protection: to prevent timing errors in tasks, Interrupt Service Routines (ISRs), or system resource locks from interfering with higher ASIL functions
- Service Protection: to capture invalid use of the OS services API by the appli-

cation

- OS Related Hardware Protection: to protect privileged hardware elements from being modified by lower ASIL functions

Note the OS supports four scalability classes with the following features:

- SC1: Deterministic Real time operating system (OSEK OS based)
- SC2: Stack monitoring and precise time control for periodic tasks
- SC3: Support for MPU/MMU to provide spatial freedom of interference
- SC4: Timing protections

Systems that require higher safety integrity levels must use higher scalability class types.

4.3.2.1 Memory Protections

AUTOSAR OS SC3 and SC4 support freedom of interference between software partitions of mixed safety criticality through the hardware-based spatial separation of memory [11]. The aim of these protections is to prevent lower ASIL software from corrupting the data of a higher ASIL software within the same system. To understand the hardware-based memory protection capabilities of automotive embedded systems, we studied the ARM Cortex M architecture which is among the most popular micro-controller architectures used in automotive ECUs [32]. Rather than an MMU, such MCU's rely on an MPU to restrict access to certain memory regions. Also, the CPU supports two modes: privileged and user mode. Only privileged mode allows access to special registers like the MPU configuration. Using the MPU it is possible to define memory regions with specific attributes such as read, write, and execute as well as specify access rights by privileged or user modes. AUTOSAR OS supports protecting memory both at the Task/ISR Category 2 level and at the OS application

level. When switching to a "non-trusted" OS application, the OS can re-configure the MPU to restrict access to the safety application code, data, and private stack. This prevents a fault in a lower ASIL OS application from corrupting the data of a higher ASIL OS application. Note in addition to MPU based stack protection, AUTOSAR OS defines software based stack monitoring which can identify that a task or ISR has exceeded a specified stack boundary at context switch time. This is done by checking a unique stack pattern which is inserted at the end of the reserved stack space. The downside of this protection is that an attacker can overflow part of the stack without crossing the stack protection boundary and evade detection. Besides data protection, the MPU can be used to restrict access to memory mapped registers to prevent certain tasks from modifying hardware registers which are safety-relevant. Dynamic MPU reconfiguration adds considerable CPU run-time overhead, therefore for most systems, a static MPU configuration is desirable. Due to the low number of MPU ranges supported in hardware, the power of the MPU as a security countermeasure is limited. Note, AUTOSAR uses the term trust in the context of safety which can be misleading because cyber security threats are not considered. Consequently, it is possible to define a "Trusted OS Application" that has access to all memory resources even though from a security point of view, that OS Application may be vulnerable to attacks.

4.3.2.2 Timing Protections

AUTOSAR OS timing protections aim to mitigate timing faults that can exist in lower ASIL software from impacting higher ASIL software. The timing protections are:

- Execution Time Protection: detects faults in Tasks or Category 2 ISRs that exceed their execution budget
- Locking Time Protection: detects faults in blocking resources, and locking in-

terrupts for a period longer than the configured maximum

- Inter-arrival time protection: detects faults in the time between successive activations of tasks or Cat2 interrupts to ensure a minimum separation time is not violated.

The OsTaskExecutionBudget is a configurable parameter that specifies the maximum allowed execution time of a task [11]. By monitoring this time, AUTOSAR OS can detect timing errors before they can lead to tasks missing their deadlines. This prevents the propagation of timing errors to higher priority tasks and allows the OS to isolate the offending task. One use case for the locking time protection mechanism is to prevent global interrupts from being disabled for a period of time that would create instability in a real-time system. Disabling global interrupts is needed in scenarios where a routine needs atomic access to a resource and cannot tolerate being interrupted. But doing so beyond a specific time threshold can prevent the real-time system from being able to process critical tasks within the required time. The third timing protection mechanism is needed to ensure the system is not being excessively interrupted which would starve the CPU from runtime cycles to perform its normal tasks.

4.3.2.3 Hardware Protection

AUTOSAR OS is expected to run in privileged mode which gives it access to special hardware registers that need protection from corruption by Tasks or Cat2 ISRs running in user mode. Example registers that are only accessible in supervisor mode can be the MPU configuration, the OS Timer unit, and the interrupt control configuration. Protecting those registers from faults in lower ASIL software is mandatory to ensure the integrity of the OS operation.

4.3.3 Watchdog Manager

AUTOSAR defines three modules for supporting watchdog functions [14]:

- Watchdog Driver: services the hardware watchdog whether internal or external
- Watchdog Interface: provides a high level of abstraction of watchdog driver functions
- Watchdog Manager: supports the supervision of multiple software entities and the triggering of an MCU reset in case of a supervision failure

Supervised entities can be software components, runnables, or Basic Software(BSW) modules that report checkpoint events to the watchdog manager. The user configures the time and sequence of checkpoints within a supervised entity. The watchdog manager then monitors aliveness, and control flow within these supervised entities. The user calls WdgM_CheckpointReached() at specific code locations to notify the WdgM that an execution event has been reached. A software error that prevents the checkpoint from being reached by the pre-determined deadline or with the right execution sequence, results in the detection of a fault by the WdgM during the execution of the WdgM_Mainfunction. The response to any such fault can range from a simple callback that notifies the user of the error, to a complete system shutdown.

4.3.4 Core Test

Safety critical applications require the monitoring of an MCU core functions to detect hardware faults during startup or normal run-time of an ECU. The core test module [5], can perform tests of MCU components such as:

- Arithmetic Logic Unit(ALU)
- Memory Protection Unit(MPU)

- Cache controller
- Interrupt Controller

The core test is executed in partial tests as a background task that can be interrupted by higher priority tasks. However, the core test requires uninterrupted execution of atomic sequences. In case a core test fails, the module reports the event to the Diagnostic Event Manager(DEM) [8], to take action based on the severity of the detected failure. Note that micro-controllers with dual lock step cores do not need the Core Test module since the dual core lock step feature can detect errors covered by this module.

4.3.5 RAM Test

The RAM test module [12], provides a physical health test of RAM cells and RAM registers to meet the fault coverage requirements of a safety critical application. The tests can either be executed in a background task or through a direct call from the application. In case a failure is detected, the module reports the results to the DEM to take the appropriate action. AUTOSAR defines interfaces to start and stop the tests but there is no direct interface to force the test status to failure. The module splits tests into atomic units that are not interruptible. A higher priority task can interrupt the module test in between atomic test units execution. The background task performs the tests of all configured blocks sequentially and repeats the sequence after each complete test is finished. One of the limitations of this module is that during the execution of the RAM test algorithm, another software shall not attempt to modify the area under test. This is to ensure data consistency in multi-core systems or with DMA controllers. The AUTOSAR specification lists the following algorithm types that are supported:

- Checkerboard test algorithm

- March test algorithm
- Walk path test algorithm
- Galpat test algorithm
- Transparent Galpat test algorithm
- Abraham test algorithm

4.3.6 Flash Test

The Flash Test module [9], provides test algorithms for non-volatile memory to meet the diagnostic coverage requirements in a safety critical system. Tests are divided into partial tests based on the number of cells tested in one task cycle. Unlike the RAM and Core tests, the Flash test can be pre-empted at any point because it does not require atomic access. It is possible to abort or suspend the flash test but that introduces a latency based on when the request is received related to the background task cycle time. A failure during the test algorithm is reported to DEM to take the proper action. The different test algorithms supported are:

- 8,16,or 32 Bit CRC
- Checksum
- Duplicated Memory
- Error Correcting Codes(ECC)

4.4 Summary of Security Gaps

By examining each safety mechanism listed above and following the SDAS approach we can extrapolate the hazards which must be prevented. Next, we determine if those hazards can be materialized through a security attack. In case the hazard

cannot be induced through a cyber attack, or the risk of cyber attack is quite low, then no further action is needed. For hazards that could be induced maliciously with a significant probability, security countermeasures are needed. First, we examine if AUTOSAR has any security countermeasures that can be used, and if not then we identify those as security gaps that require further handling. For AUTOSAR countermeasure, it is necessary to identify the residual safety impact on the software architecture which will require further system refinement. The results of this analysis are summarized in Table 4.3. Let's walk through the analysis of hazards corresponding to "OS: Memory Protection for FFI". The non-malicious hazard is that the data of a safety application is corrupted by a fault in another application causing the safety application to misbehave in a serious way. An adversary can materialize this hazard by injecting malicious code that can tamper with the safety application data causing an unsafe action. To mitigate this type of threat using AUTOSAR, one has to treat all applications as untrusted to ensure that they all run in user mode. As a result, the OS is given the duty of switching contexts and enforcing spatial separation through the MPU. But implementing such security countermeasure has impacts on the runtime performance of the CPU due to increased switching time. This requires further impact analysis to harmonize safety and security requirements. By following this systematic approach, we can determine security gaps or deficiencies within AUTOSAR. These will be the driving factors of the solution presented in the next chapter.

Safety Mechanism	**Threats**	**AUTOSAR Countermeasures**	**Other Countermeasures and Residual Safety Impact**

E2E: CRC	Malicious data manipulation	SecOC: Authenticate PDU	Authentication time shall not result in delay of signals for proper data handling, therefore, use hardware acceleration and optimized software to authenticate fast
E2E: sequence counter	Malicious sequence counter manipulation	SecOC: Authenticate sequence counter	Same as above
E2E: alive counter	Malicious alive counter manipulation	SecOC: Authenticate alive counter	Same as above
E2E: IPDU ID	CAN ID spoofing	SecOC:Authenticate CAN ID	Same as above
E2E: timeout detection	DoS attack to prevent data reception	None	With IDS, disabling the message relays of offending nodes shall be done in a safe way to prevent false positives from causing unintended shutdown of safety related functions that rely on the missing messages

OS: Memory Protection for FFI	malicious app corrupts the data of a safety application	Treat all applications as untrusted and let OS configure MPU dynamically	Evaluate MPU context switching time on real-time performance and availability of safety application
OS:Timing Protection	Run-time exhaustion attack to cause certain tasks to run longer than they should	Configure Task Execution Budget to factor external threat sources	Implement external component that can monitor the timing execution without the possibility of being spoofed. Attack detection response shall be harmonized with safety constraints
Watchdog Manager: flow control	Code injection or reuse attack to alter the control flow	Configure supervised entities to protect security related functions	External component that can enforce CFI checks without the possibility of being disabled. Attack detection action shall be harmonized with safety constraints

Flash Test: memory faults	Attacker tampers with code or data in flash	None	Authenticate code and critical flash data to detect malicious tampering. Time for data authentication shall not violate the availability needs of the ECU

Table 4.3: SDAS results against AUTOSAR Safety Goals

4.5 Exploiting Security Gaps

To demonstrate the need for improving AUTOSAR security, we identify security gaps from the above analysis and build two practical attacks. With the E2E library, the attacker cannot spoof the data when authentication is enabled, but he can create a message loss scenario by flooding the network with high priority CAN messages. With AUTOSAR OS timing protections, the attacker can violate the CPU timing budget of safety functions by overwhelming the system with network authentication requests.

4.5.1 Message Loss Attack

In order to create a message timeout event, the attacker can flood the bus with high priority CAN messages, e.g. zero-ID CAN messages, at the highest periodicity possible for the target baud rate. Transmitting zero-ID frames in a back to back fashion will reduce the likelihood that a valid frame wins arbitration to be transmitted on the bus. As a result of transmitting nodes continuously losing arbitration to the zero-ID message, receiving nodes will start logging timeout faults. Subsequently, control functions that rely on those messages will be degraded, which is the safe state

of missing safety critical messages. An attacker determined to prevent the safety critical ECU from performing its intended function can successfully launch this attack by exploiting this mechanism. Although the system detects the attack, the attacker's goal of shutting down safety functions is still attained.

4.5.1.1 Security Countermeasures

In [81], we proposed several countermeasures to the message loss scenario. A smart gateway that runs intrusion detection software can monitor the received CAN message identifiers along with their expected frequency and detect attacks such as the zero-ID flood attack. Alternatively, security monitoring software within the transmitting ECU can detect the malicious manipulation of the CAN configuration. The monitor can either reside in the HSM or operate in privileged mode with protections from the HSM to ensure it is not disabled. If the CAN settings are flagged as tampered, the HSM can leverage its dedicated IO pins to disable the CAN transceiver and prevent the disturbance of the local network. Since this attack is not fully preventable, it is necessary to collect indicators when timeout events occur in order to distinguish normal failures from security attacks. This can be achieved by logging the frequency of these failures, as well as capturing additional network traffic to aid in the anomaly detection either through a local or off-board intrusion detection system. Although the zero-ID attack has already been mentioned in other publications, such as [73], the attack is still worth mentioning here because we arrive at by applying the SDAS approach.

4.5.2 Task Execution Budget Attack

By considering the OS timing protections and how they can be violated maliciously, we arrive at our second attack method which derives from the Task Execution Budget monitoring. As mentioned in Section 4.3.2.2, AUTOSAR OS monitors the

task execution time, to prevent a single task from starving the CPU from run-time resources. In response to this type of timing error, the OS defines the following possible actions that the application can request[11]:

- PRO IGNORE: the OS can ignore the event
- PRO TERMINATE TASKISR: the OS shall forcibly terminate the task
- PRO TERMINATE APPL: the OS shall terminate the faulty OS Application
- PRO TERMINATE APPL RESTART: the OS shall terminate and then restart the faulty OS Application
- PRO SHUTDOWN: the OS shall shutdown itself

Upon detecting the error condition, the OS triggers a ProtectionHook to notify the application to take fail-safe measures. The last action from the above list implies that the system can be completely shut down as a result of such an error condition. AUTOSAR OS gives the system configurator the flexibility to specify the appropriate value for the execution budget as well as the proper behavior in case it is exceeded. In a stable system absent from a malicious attacker, such a fault is normally caught during development when the system is tested under maximum load conditions. However, in the presence of an attacker who is able to repeatedly cause this error condition, the system can experience constant resets that prevent it from ever being able to execute its intended safety functions. To realize the attack goal, the attacker has to cause an OS task to exceed its execution budget. One way to find candidates for this type of vulnerability is scanning the application for processes that have variable execution time due to their dependence on a hardware resource like flash programming time, or a network resource. We chose the latter and we investigated how to exploit CAN networks that support authenticated messages via the Secure On-Board Communication (SecOC) module [10]. When a secure Protocol Data Unit (PDU) is received,

SecOC receives an indication from the Protocol Data Unit Router (PDUR) module to copy the PDU to its own memory buffers. It then triggers the verification of the authenticator portion of the PDU by calling the AUTOSAR Cryptographic Service Manager (CSM) module as illustrated in Figure 4.1. Only if the verification passes, SecOC then notifies the PDUR module to route the PDU up to the consuming layers [10]. Since SecOC relies on the SecOc_MainFunction() to perform the verification processing, the attack goal is to cause that function to exceed the AUTOSAR configured run-time budget: OsTaskExecutionBudget. Based on the CAN FD specification [55], we can estimate the nominal time for transmitting a CAN frame if the arbitration rate, data rate and payload size are all known. As shown in Figure 4.3, the number of bits in a CAN FD frame can be calculated based on the different segments of the frame. Note, the length of the CRC field is either 17 bits or 21 bits depending on the payload size. For simplification, we set the CRC field to be 21 bits which corresponds to a payload length of 20 and 64 bytes. This choice is guided by the fact that in a vehicle CAN FD frames are more likely to utilize the larger payload size. Thus the only unknown variable parameter remaining is the number of stuff bits which depends on the content of the CAN frame. The rule is that no more than 5 bits can be transmitted consecutively with the same polarity. Therefore, stuff bits are inserted to ensure bit polarity is toggled if more than 5 consecutive bits have the same logic level. Accounting for all the variables, results in a formula that gives us the estimated transmission time of a CAN FD frame (in seconds):

$$T_{canfd} = (1 + f) * (\frac{30}{a} + \frac{(28 + dl * 8)}{d}) \tag{4.1}$$

where a is the arbitration baud rate in bits per seconds, f is the stuff bit factor, d is the data baud rate in bits per seconds, and dl is the frame data length in bytes. Note that in a worst case scenario, 1 stuff bit is inserted for every 5 consecutive bits which

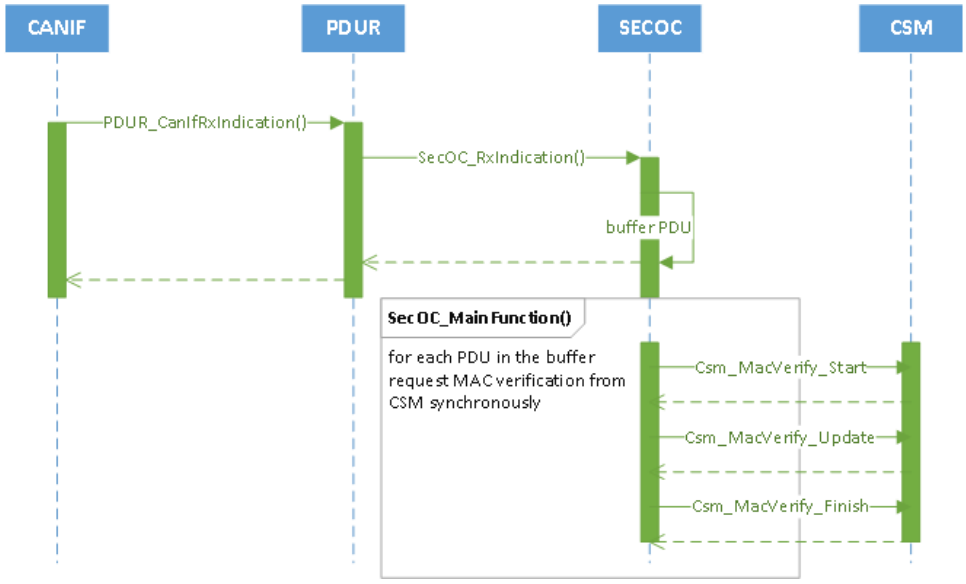

Figure 4.1: Data flow from CANIF to the CSM layer for frame authentication

is equivalent to a factor of 20%. When CAN message authentication is enabled, the attacker takes advantage of the fact that an ECU needs to spend a fixed amount of CPU run-time to perform a MAC authentication before the frame is accepted or discarded. Note, an attacker does not have to worry about generating valid MAC values, because the goal is to exploit the time taken to verify the MAC, not to spoof a message with a valid MAC. The processing time varies depending on the target microcontroller and the CPU operating clock frequency. SecOC defines a parameter for the number of authenticating attempts when the freshness counter is not transmitted in its entirety within the frame. The parameter: SecOCFreshnessCounterSyncAttempts, causes the re-authentication of a secured I-PDU with different freshness values within the acceptance window until one authentication succeeds or all attempts fail. This results in more processing time for each message authentication failure. Therefore, this parameter shall be accounted for in the attack potential evaluation. As shown in Figure 4.1, SecOC_MainFunction() loops through all the buffered PDUs that require verification and triggers the verification request to the AUTOSAR CSM [6] module. We intentionally choose to configure CSM to run in synchronous mode so as to maxi-

mize the processing time spent in SecOC_MainFunction as it tries to authenticate all frames in the buffer before the task is finished. As a result, SecOc_MainFunction() has to wait for the three CSM steps to be completed before it starts processing the next secure PDU. To achieve a successful attack, the attacker needs to send a burst of authenticated PDUs that would result in the SecOc_MainFunction() exceeding its run-time execution budget as shown in Figure 4.2. The key here is finding the minimum size of the frame burst needed to cause the timing error condition and then checking whether it is feasible given the constraints of the CAN FD protocol. The attack is possible if $\exists$ a value $B \leq maxB$ such that $T_{processing} > T_{budget}$ where:

$$maxB = \frac{T_{secoc}}{T_{canfd}} \tag{4.2}$$

Therefore, assuming SecOc_MainFunction has a task cycle time of T_{secoc} and a CAN FD frame transmission time of T_{canfd}, our goal is to find the minimum burst size B such that the processing time of SecOC_MainFunction, $T_{processing}$, is greater than the configured execution budget T_{budget} while $B \leq maxB$. In order to evaluate if a system is affected by this attack, we present Equation 4.3, for calculating burst size B. Let T_{mac} be the MAC verification time for verifying a single 64 byte message, note this time depends on the MAC algorithm and whether it is accelerated in hardware or implemented in software. Let T_{main} be the run-time to execute the SecOC_MainFunction() to process a single frame without the MAC calculation overhead. Let T_{budget} be the maximum execution budget of the SecOC_MainFunction task. Let $N_{attempts}$ be the value of SecOCFreshnessCounterSyncAttempts, which is the number of attempts performed if MAC verification fails. Let B be the number of CAN FD messages that can be verified within the T_{budget} time, then:

$$B = \frac{T_{budget}}{N_{attempts} * (T_{main} + T_{mac})} \tag{4.3}$$

The above analysis gives us the conditions needed to determine if the mechanism can be triggered externally based on the target system parameters. An evaluation on a real target is shown in section 4.5.3.

4.5.2.1 Security Countermeasures

The countermeasures to this attack as shown in [81], require several steps. First, when defining Task Execution Budgets, system designers shall take into account potential security threats to the task execution time to find the optimal budget that can address both safety and security-related faults. Moreover, wherever possible, the asynchronous mode for performing specific functions shall be chosen. AUTOSAR provides both synchronous and asynchronous mode in several modules like NVM and CSM. This would limit the time spent in a task as it allows the job to be performed over several call cycles. With CAN authentication, SecOC defines a maximumretryCounter to repeat the MAC verification with a different freshness counter upon failure. This can further exacerbate the duration of performing the CAN authentication when an attacker sends a burst of invalid messages(wrong MAC). Therefore, using the minimum value possible for this parameter is recommended. Alternatively, a network anomaly detection system can flag and potentially stop anomalous message bursts like the one presented here to trigger the attack. Note, the effectiveness of an IDS with this attack may be limited because message bursts do occur naturally. Finally, a fatal error such as a system shutdown due to exceeding the execution budget may seem highly improbable under normal conditions, but under the influence of an attacker can become much more likely. Therefore, it is necessary to review all fatal errors in the application to re-evaluate if the conditions of such errors are impacted by a malicious attacker. In the cases where the attack cannot be prevented, it is necessary that the fail-safe response collects additional indicators to aid in future forensic analysis. Some of those indicators can be the network traffic at the time the

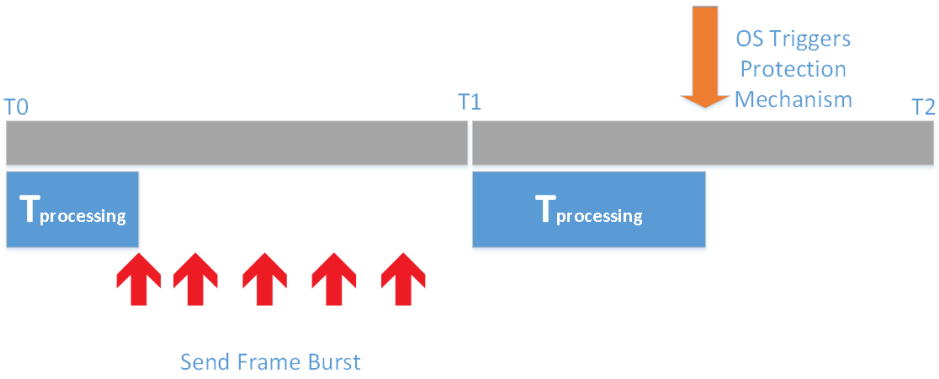

Figure 4.2: Triggering the OS protection mechanism

fault occurred, as well as the frequency at which the error condition occurred. This information can be analyzed by a vehicle IDS or an offline system that can observe inputs from many vehicles to identify fleet wide attacks.

4.5.3 Demonstrative Attacks

To demonstrate the attacks presented in section 4.5, we build a test environment using a Renesas RH850 32-bit micro-controller, operating at a 120 Mhz CPU clock as the test target, and Vector CANalyzer as a simulated attacker. The two are connected together through a CAN FD link with an arbitration baudrate of 500 Kbps and a data rate of 2 Mbps.

Zero-ID Flood Attack: To simulate the attack, CANalyzer is used to send messages on two CAN FD channels connected together into a single CAN FD channel on the target board. The micro-controller target board controls a servo motor by translating the CAN messages into PWM signals that control the steering and driving. This is representative of a malicious attacker that has direct access to the CAN bus where the target ECU resides. The aim is to observe the impact of the zero-ID flood attack on the ability of the target board to steer or drive the car. On the flood path channel, a Communication Application Programming Language (CAPL) script is used to send the zero-ID message in a back to back fashion. On the Normal

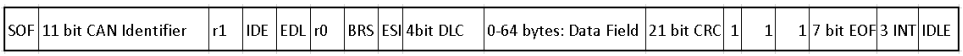

Figure 4.3: CAN FD Frame layout, for 64 byte frames CRC is 21 bits long

channel, a CAPL script simulates the drive and steer messages which cycle through a sequence of steering and throttle messages. By enabling the flood attack, control of the servo motor becomes very difficult as only a small fraction of control messages are transmitted on the bus. In a real vehicle with timeout monitoring protection, the receiver would simply disable functions that require CAN data within the steering and power train ECUs which practically corresponds to a successful DoS attack.

Resource Exhaustion Attack: CAN FD extends the CAN 2.0 standard with a larger payload (up to 64bytes) and a higher data rate (up to 8 Mbps) [55]. The protocol defines an arbitration baud rate, and a flexible data rate that can be higher than the arbitration rate. This allows CAN2.0 frames to coexist with CAN FD frames. In order to evaluate this attack we implemented a simplified CAN driver along with a minimal SecOC component that performs the entire chain of CAN message reception and authentication. We assumed that the AUTOSAR CSM [6], is configured in synchronous mode, as a result, SecOC_MainFunction() waits until a buffered secure PDU is authenticated before triggering the next one as shown in Figure 4.1. The process is repeated until all the buffered secure PDUs have been verified. The attacker was simulated by a software task that runs every 10ms and produces a variable number of CAN FD messages within a single burst on CAN channel 2 of the micro-controller. A CAPL script in CANalyzer relays the messages from CAN channel 2 to CAN channel 1 to trigger the authentication in the SecOC_MainFunction(). We configured the receiver to process the CAN messages on CAN channel 1 in a 10ms cyclic function, i.e. $T_{secoc} = 10ms$. The CAN controller was setup to receive a maximum of 40 unique messages on the same channel. We also set the SecOCFreshnessCounterSyncAttempts value to 1, because the freshness counter

Arbitration Rate	500 kbps
Data Rate	2000 kbps
Data Length	64 bytes
Arbitration Time	39.1μs
Data Time	314.4μs
Total Frame time	359.5μs
SecOC Cycle Time	10 ms
Burst Size	27.81 frames

Table 4.4: CAN FD frame time in μs based on 64 byte DLC

is sent in its entirety within the CAN FD frame. This is also meant to increase the attack difficulty, because a larger SecOCFreshnessCounterSyncAttempts increases the $T_{processing}$ time needed to verify all the failed MAC values received. Due to its prevalence in embedded systems, we choose the AES-128 CMAC as the authentication algorithm. Thus the CAN FD frame was constructed to contain 48bytes of payload data, 8bytes freshness counter and 8bytes truncated CMAC. As for the stuff bit factor f, we chose a factor of 15% which is below the maximum value and more biased towards the worst case condition. We then toggle a port pin around the function SecOC_MainFunction() to measure $T_{processing}$ with an oscilloscope. Using Equations 4.1 and 4.2, we can determine that for the parameters of our experiment outlined in Table 2, the maximum burst of messages possible to attack the system is 27 messages with a payload of 64 bytes each. By choosing a 64byte CAN FD frame length we aim to increase the attack difficulty by minimizing the maximum burst size possible within our time constraint of 10ms. The next step then is to find the burst size for which $T_{processing}$ exceeds T_{budget}. Arriving at the execution budget is highly dependent on the application and how the operating system is configured. Typically, the system designer chooses the execution budget of individual tasks based on a static analysis aided by tools that can estimate worst case execution time. For our evaluation since we do not have a real application we assume that SecOC_MainFunction will be among several cyclic functions that are part of the 10ms Task. Thus we choose the T_{budget} to

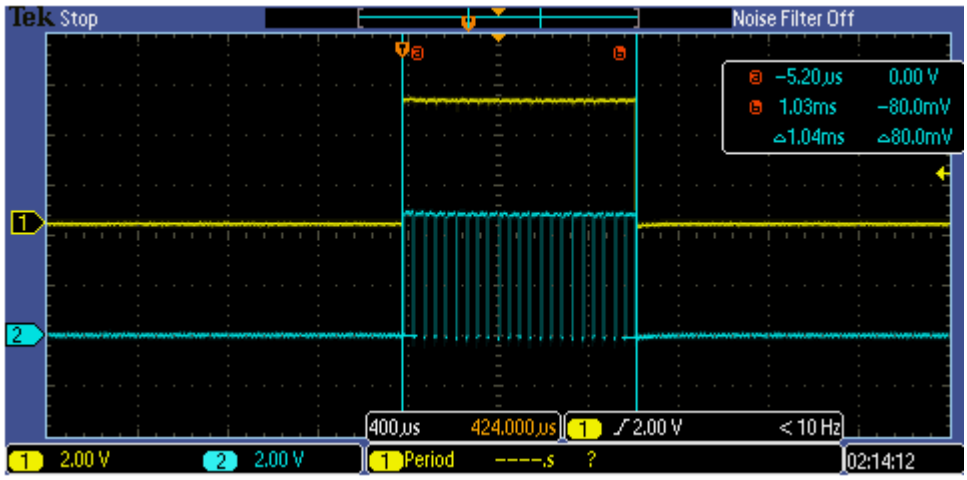

Figure 4.4: 1.04ms of total run-time for processing 22 CAN FD messages

be 10% of the task cycle time which corresponds to 1ms. In a real system, execution times can be better estimated based on the demands of the target application. The results of our experiment in Figure 4.4, show that for $T_{processing}$ to exceed our chosen execution budget of 1ms, it is sufficient to send a burst of 22 secure PDUs within a 10ms cycle. Therefore the number of frames needed to trigger the AUTOSAR OS failure is below the $maxB = 27$ calculated in Table 4.4, which satisfies our condition for a successful attack. Furthermore, 22 CAN messages is well within the normal number of CAN frames that a typical ECU consumes.

4.6 Extending Safety Mechanisms for Security

The previous section showed how analyzing the safety mechanisms allowed us to identify security gaps in AUTOSAR based systems which needed to be addressed. Now we turn our attention to the question of whether we could use AUTOSAR built-in safety mechanisms to improve the security posture with minimal impact to cost and legacy software. Since such protections were not designed to cope with security attacks, we provide additional constraints to re-purpose those features to cope with security threats. Note, a previous study by Bohner et al. [34], highlighted several of

the same protections, but we extend those to include remaining deficiencies to set the stage for the work in the next chapter.

4.6.1 Stack Usage Monitoring

Code injection attacks through buffer overflow vulnerabilities [49], continue to be among the most prevalent and effective attacks in computer systems. An attacker overflows a buffer boundary to load a malicious payload into the victim's memory. This enables rerouting the flow of the program to the new address overwritten by the malicious payload. Naturally, a protection mechanism that can detect stack overflow would be relevant for security. As mentioned in Section 4.3.2.1, AUTOSAR OS supports memory stack overflow monitoring either through software checks (by checking special patterns on the stack) or with the help of the MPU. Since a malicious attacker can easily forge the stack pattern value after overwriting the stack space, software-based stack protection cannot be considered a robust security mechanism. With the MPU based stack protection, the OS sets up a dedicated MPU stack entry prior to activating the corresponding task. The user defines the stack size for each context based on prior measurements to determine the maximum required stack size. An attacker who manages to inject code in a stack space cannot exceed the stack boundary and cannot inject code in a stack dedicated for another context. Doing so results in an immediate exception which results in the OS taking corrective action such as issuing a reset.

4.6.2 RAM Execution Prevention

Modern operating systems support data execution prevention (DEP) to ensure that a memory address can either be configured as writable or executable but not both. Automotive embedded operating systems do not explicitly support such protections, but using AUTOSAR OS, it is possible to emulate this protection. As

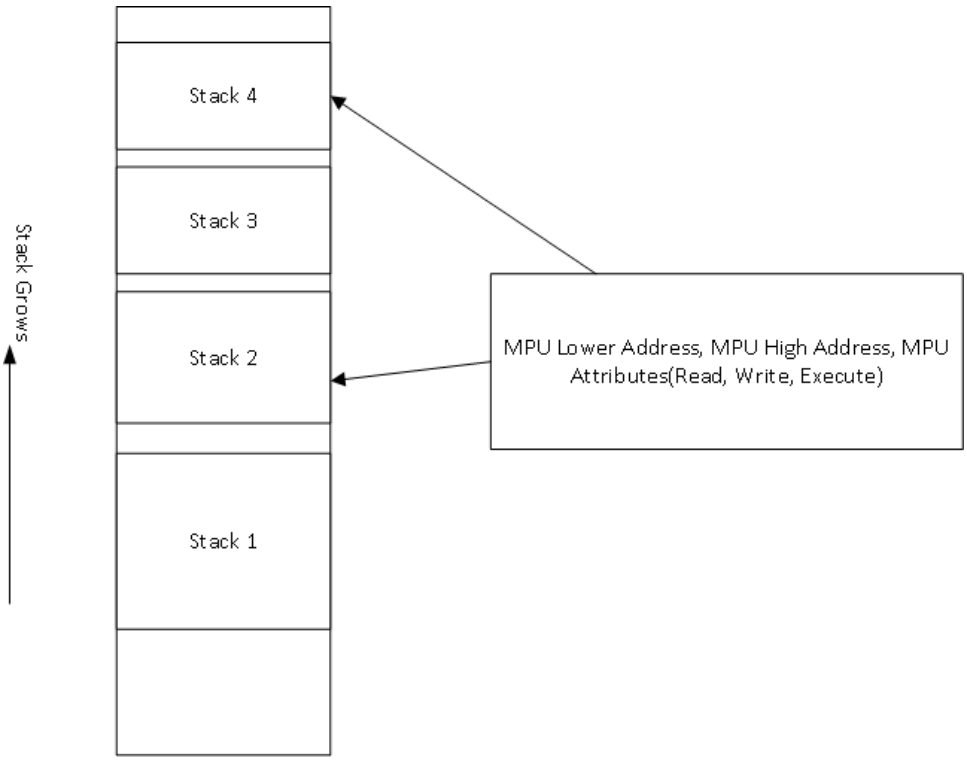

Figure 4.5: MPU based stack protection

mentioned in Section 4.3.2.1, using the MPU, it is possible to set up access rights to memory regions as: read, write, and execute. While AUTOSAR OS does not explicitly define how to separate access rights, it is expected that specific vendor implementations of AUTOSAR OS offer the user the ability to assign different access rights to different OS applications. We propose extending this to restrict all execution out of RAM regardless of the safety level of the corresponding application. While this does not prevent all stack-based attacks as is the case with a return-to-libc attack [47], it does raise the difficulty level for mounting attacks on embedded systems. An attacker who attempts to violate this rule will immediately cause a CPU exception which can reset the system to restore it to a known secure state.

4.6.3 Flow Integrity Protection

The topic of control flow integrity(CFI) was first introduced in [24]. Moreover, [51], showed several techniques to alter the execution flow specifically in an embed-

ded system. Having a mechanism that can enforce program flow can be useful in mitigating attacks such as return oriented programming through stack-based code injection. AUTOSAR offers a mechanism that can be re-purposed for CFI, namely the checkpoint monitoring in the Watchdog Manager(WdgM). Using the WdgM, it is possible to define supervised entities(SE) which are code elements that can be monitored for the order of execution. This results in creating an internal graph of code segments that shall be executed in a specific sequence with time constraints between checkpoints. In Figure 4.6, we show an example where checkpoints are inserted to allow the WdgM to enforce the program flow of a password checker. If an attacker manages to jump to CP1-2 to unlock the security state, the WdgM will detect a program flow violation because CP1-0 and CP1-1 were bypassed. During the execution of WdgM_MainFunction(), the WdgM detects the violation and can trigger a watchdog reset by not refreshing the watchdog timer. If the attacker chooses to reroute control to his own routine and disables the calling of the WdgM_MainFunction(), the watchdog timer will also trigger a reset because it has not been serviced. While WdgM can increase the difficulty of control flow attacks, it suffers from two weaknesses. An attacker can spoof checkpoint events by calling the routine that reports the checkpoint entry. Also, an attacker can refresh the watchdog on his own to prevent a watchdog reset. In the next chapter, we will see how we address these problems with the HSM based security monitor.

4.6.4 OsTiming Protections

The interrupt locking time protection can be useful in detecting attacks in which a malicious application attempts to disable interrupts for an extended period of time to complete an attack. The inter-arrival time protection can be useful to detect DoS attacks in which an attacker attempts to overwhelm the system by triggering an interrupt too many times to starve the CPU of run-time cycles or to exhaust stack

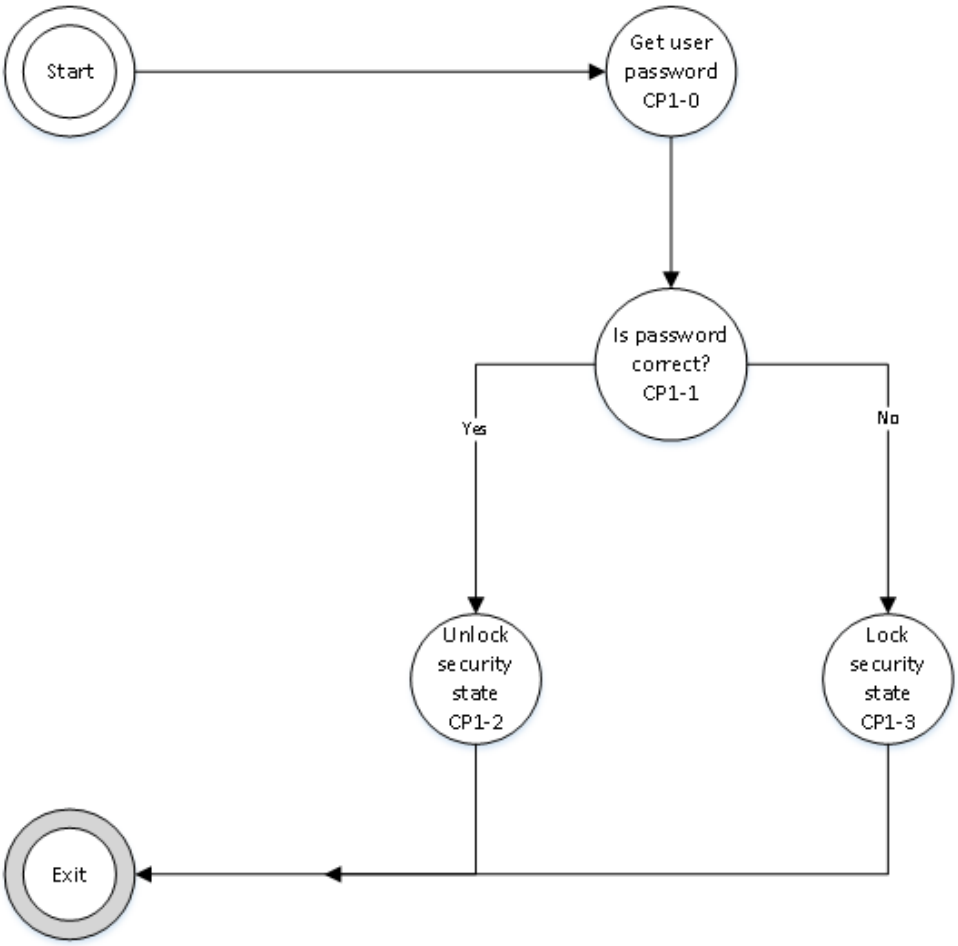

Figure 4.6: WdgM monitors password verification checkpoint

memory resources if nested interrupts are allowed. An example would be malicious network traffic that results in over triggering the CAN interrupt.

4.6.5 Hardware Resource Protection

AUTOSAR OS relies on the MPU to prevent lower ASIL applications from corrupting the data of a higher ASIL application including register settings. One potential use case for security is prohibiting access to the CAN registers by mapping them to a protected MPU entry. This prevents malicious code from directly interacting with the CAN controller to spoof the CAN bus through an infected ECU. This limits access to the CAN controller registers to the CAN driver, but this comes at the cost of

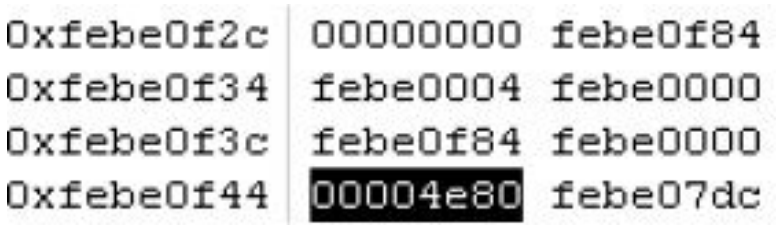

Figure 4.7: Stack content before attack:return address = 0x4e80

increased CPU overhead to switch contexts and reconfigure the MPU when accessing the CAN driver.

4.6.6 Demonstrative Protections

Similar to the attack setup, we reuse our RH850 micro-controller and this time the GreenHills debugger environment to evaluate the defenses shown below.

4.6.6.1 Stack Overflow Protection

To demonstrate this security mechanism we create a CAN diagnostic vulnerability that allows injecting code into the stack memory reserved for a "trusted OS application". The diagnostic routine that is loading the data over CAN contains a bug that allows overflowing the stack with a well-crafted diagnostic request payload. As a result, the return address of the calling routine is overwritten with a RAM-based address which corresponds to the entry point of the malicious routine. Since we did not have access to a full AUTOSAR software stack at this point, we implemented a minimal RTOS that is responsible for setting up the MPU to protect the stack memory against writes by unauthorized software partitions. We also disabled execution rights for stack based memory to prevent an attacker from directly executing injected code. Figure 4.7 shows the stack state before the attack with the target address to overwrite being 0x4e80. The code listing shown in IV.1, is a simple routine that contains the used buffer overflow exploit.

Listing IV.1: Routine containing buffer overflow vulnerability

```
0xfebe0f34 | 00000000 55aa55aa
0xfebe0f3c | 55aa55aa 55aa55aa
0xfebe0f44 | febff000 febe07dc
0xfebe0f4c | febe0fdc 000400db
```

Figure 4.8: Stack content after the attack: return address = 0xfebff000

```
void maliciousRoutine(void)
{
        reprogrammingFlag = C_TRUE;
0xfebff000  maliciousRoutine:       0a01            mov     1, r1
0xfebff002  maliciousRoutine+0x2:   0f4480df        st.b    r1, -32545[gp]
        asm("jr 0x1000"); // jump to the bootloader
0xfebff006  maliciousRoutine+0x6:   07801000        jr      0xfec00006
}
0xfebff00a  maliciousRoutine+0xa:   007f            jmp     [lp]
```

Figure 4.9: Malicious routine is successfully entered after the stack overflow

```
void ApplDiagWriteDataByIdentifier ( uint8 canLen ) {
uint8 diagBuffer[8] ;
memcpy( diagBuffer , canBuffer , canLen ) ;
}
```

The routine copies data from CAN to the diagnostic buffer without checking if the request length fits into the receive diagnostic buffer. Since the buffer is a local variable, the stack is overrun and the return address of the routine is overwritten. Figure 4.8, shows the contents of the stack after executing the routine which causes the overwrite of the return address with the target routine address: 0xfebff000. The latter contains a jump into the bootloader in order to bypass application security checks and initiate flash programming with a new malicious firmware image. Once the routine attempts to return, the CPU pops the link pointer register value *lp* from the stack causing it to jump to the malicious code which then jumps to the bootloader, as shown in Figure 4.9. Due to the MPU security mechanism being active, the vulnerable diagnostic routine is only able to overflow its own stack, but not the stack of other OS applications. Although the stack overwrite is possible, the attempt to fetch the code to jump to the bootloader immediately triggers an exception as shown in Figure

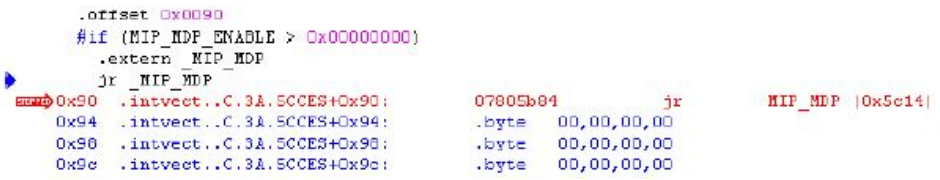

Figure 4.10: MPU exception triggered due to violation of execution rights

4.10. Once the exception is triggered, the system can log the address where the violation occurred and store that in non-volatile memory for later intrusion analysis. The next action is to shut down safety functions and reset the system to restore the CPU back to its original state. Note, that action has to take into consideration the vehicle state to avoid creating a sudden loss of a safety function which would lead to a safety goal violation. In order to clear the malicious code, it is highly recommended that the CPU always resets the contents of the RAM and stack during startup.

4.6.7 Recommendations for AUTOSAR Protections

Before we can rely on AUTOSAR safety mechanisms for security protection, the software authenticity must be checked at startup by performing secure boot to establish trust in the AUTOSAR software layers. In terms of the watchdog manager being used to protect entry into certain critical routines, the global state variables of the watchdog manager need protection against tampering. Since the watchdog manager like other basic software services is executed in user mode, it is vulnerable to a malicious application that directly modifies its variables. Therefore, it is recommended to treat the watchdog manager as a trusted application that is executed in supervisor mode. Moreover, the term: "Trusted OS Application" is a pure safety characterization which does not imply any security assurance. Thus we recommend that CPU privilege mode is reserved strictly for the OS and a handful of basic software services that control security, while all other tasks and applications run in user mode. The software shall be partitioned not only based on safety criticality but also

based on security relevance. Having separation between security relevant software and non-security relevant software further enforces the principles of security isolation and least privilege. This does come at the cost of reduced performance due to the limitations of the MPU to handle a large number of memory partitions. In order to raise the difficulty of code injection attacks that can disable protection mechanisms, RAM-based execution shall be disabled via the MPU for all RAM partitions(data and stack) in both user and privileged modes. This means that even if an attacker manages to load code on the stack, attempting to execute that code shall result in an MPU exception. The timer interrupt upon which the OS is relying shall be solely controllable by the OS. This ensures that the OS can execute the security monitoring functions defined in this chapter. When using the WdgM for control flow protection, the underlying hardware watchdog timer shall be configurable only once after reset. Attempts to disable the watchdog shall either be ignored or result in a system reset. This prevents an attacker from disabling the watchdog to prevent it from interfering with his malicious execution.

4.7 Conclusion

In this chapter, we applied the SDAS approach to AUTOSAR to uncover security gaps and recommend countermeasures. We then evaluated safety mechanisms as a means to improve the security of AUTOSAR. Several mechanisms were shown to be effective in preventing attacks but exhibited limitations in terms of performance and security guarantees. For those reasons, it became clear that additional security countermeasures were needed that would relieve the host CPU from the additional processing burden while improving the security posture of AUTOSAR based systems. Additionally, it became clear that usage of a real AUTOSAR stack was necessary to evaluate security countermeasures and their impact on legacy code. This will be explored in detail in the next chapter.

CHAPTER V

SecMonQ: Security Monitoring of AUTOSAR Based Systems

5.1 Introduction

Up until this point, our evaluation of AUTOSAR was based on analyzing the open standard specifications and attempting to emulate certain functionality through our own drivers. In this part of the dissertation, we managed to transfer our test environment to a real AUTOSAR stack provided by Elektrobit [2]. The study of AUTOSAR security gaps and protections led us to the conclusion that an additional component was needed to supplement AUTOSAR safety mechanisms. Thus, we transition to the problem-solving phase of the dissertation in which an HSM based security monitor is presented as the missing link for attack detection and/or prevention in safety-critical ECUs.

Generally, safety monitors are considered a standard feature in safety-critical automotive systems. In hardware, a safety monitor can check for deviations in core voltage or clock frequency. In software, safety monitors check for integrity of peripherals such as CAN registers. With the introduction of cyber threats against automotive systems, existing safety mechanisms and monitors are inadequate to cope with malicious fault injection and data tampering [101]. Security best practices for

automotive systems, [25], define several security countermeasures against external attacks such as locking JTAG, authenticating CAN communication [10], implementing code signing, etc. While these best practices raise the bar for a successful attack significantly, they cannot fully eliminate the possibility that an ECU is hacked and used to spoof the rest of the vehicle. Surveying published attacks against automotive ECUs, [73],[74],[98],[66], [64], they almost always follow the pattern of gaining access of some external facing interface in order to modify the firmware of an intermediate ECU that can then transmit malicious CAN messages. The target is usually to spoof a vehicle dynamics ECU like steering or braking in order to cause an unsafe driving situation. But, whether an attacker can replace firmware such as in [74], or jump into the CAN transmit service to spoof the CAN bus as in [66], safety systems must be resilient enough to detect such attacks and take fail-safe action. When it comes to vehicle intrusion detection and prevention, the bulk of the research is on CAN-based intrusion detection [58]. The focus of such systems is to create a ground truth model of the CAN traffic and then detect anomalies when the CAN data deviate from the model [93],[75], [96]. The difficulty of making such solutions practical is that vehicle data is highly dynamic in nature and what might seem like an anomaly in one model, could be a perfectly normal driving scenario in another model. There are certainly effective features for CAN intrusion detection, such as monitoring CAN message frequency, and CAN message ID lists. But even when a CAN intrusion detection system performs well in detecting anomalies, the recovery action can be quite difficult. Such action ranges from a gateway ECU IDS that stops relaying messages from the offending bus, to adding specialized hardware that can interfere with the CAN transmission of the offending ECU by injecting CAN errors [20]. Alternatively, CAN hardware fingerprinting solutions [30], [41], create an electrical signature of each ECU. This allows the receiving node to cross-check the CAN message ID against the received electrical signature at the physical layer to detect the spoofed message. The

approach shows promising results but noisy vehicle bus characteristics and impacts of electronic component aging can make this approach unreliable in the long run. In this chapter, we take a different approach to CAN injection attack prevention by adding a security monitor at the ECU level to prevent the attacker from reaching the vehicle bus and injecting hazardous messages. We leverage the built-in Hardware Security Module(HSM) as the execution environment for our security monitor to detect and recover from internal attacks that can result in spoofing other ECUs. Our initial research in this area [78] was focused on detecting code reuse attacks that bypass diagnostic security checks to execute restricted diagnostic services. In a later work [80], we extended the security monitoring system to four components to detect and prevent the most severe attacks against safety-critical automotive systems. For the rest of this chapter, we refer to the enhanced security monitor as SecMonQ. This security monitor runs on the embedded HSM in an isolated secure execution environment. The first monitoring component is a firmware integrity checker. In addition to performing a secure boot measurement at startup [84], the integrity monitor performs periodic boot authentication measurements to detect firmware manipulation during run-time. The second component is a communication peripheral integrity checker. It is responsible for periodically reading out the register configuration of peripherals such as CAN and performing an integrity check to ensure that certain fixed settings such as transmission identifiers are not maliciously modified. The third component is a time integrity monitor that checks for anomalous behavior in the real-time execution of safety-critical tasks. The fourth component is the logical flow checker which checks for anomalies in the control flow of safety-critical paths of execution. When an attack is detected, the security monitor interrupts the host, forcing it to enter a trusted failure recovery state which can then bring the system back to a safe state. In parallel, the security monitor locks security assets and logs the event for further forensics. The outcome of this work produced a novel approach to defeating the CAN

masquerading attack at the ECU level without the need for specialized hardware. Furthermore, it enabled us to present a harmonized approach for security detection and safe recovery while keeping the solution compatible with existing AUTOSAR based software.

5.2 Attack Goal and Threat Model

Our attacker's goal is to manipulate the safety functions of the vehicle to cause an unsafe driving situation similar to the Jeep hack attack [74]. He achieves that goal by modifying the firmware of the target ECU or an intermediary ECU that is then used to launch a CAN masquerading attack on the rest of the vehicle. To establish a presence in an ECU, he finds a vulnerability or a hidden backdoor that allows either the replacement of the ECU firmware in flash or, the injection of code in RAM. As a result, he is able to modify the control flow to enter safety-critical functions, or modify the CAN peripheral settings to launch a CAN masquerading attack on other ECUs. SecMonQ aims to tackle the attack patterns of the Jeep and Tesla hacks. In the case of the Jeep hack [74], the firmware of the vehicle communication processor in the UConnect unit was replaced to launch a CAN spoofing attack on the vehicle dynamics ECUs. In the Tencent Labs attack [66], the Cantx service in the Tesla autopilot ECU was reached via a security vulnerability in order to construct a data steering control message to spoof the electronic power steering ECU.

5.3 Background and Related Work

5.3.1 Control Flow Integrity

SecMonQ borrows several ideas from the mature field of control flow integrity(CFI) and adapts them for the real-time constrained nature of automotive ECUs while ensuring the safe and secure response to a control flow violation. Buffer overflow

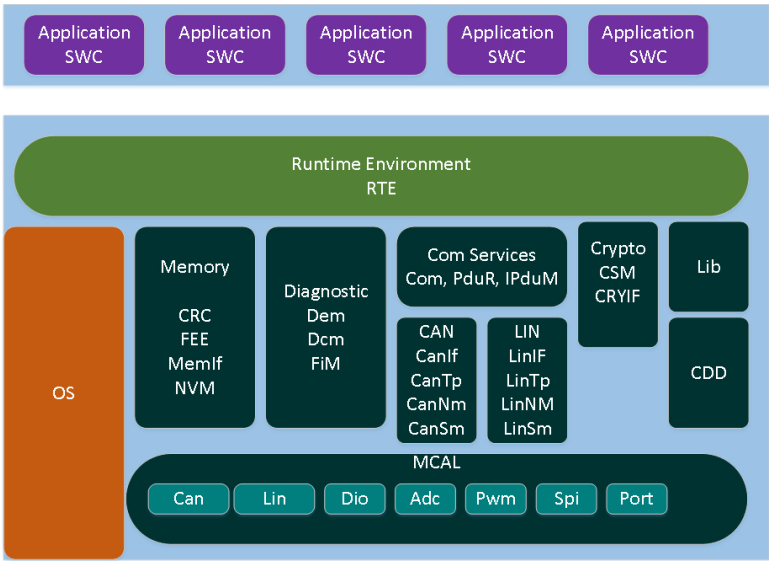

Figure 5.1: AUTOSAR Software Architecture

vulnerabilities which are among the most widely used software errors to overwrite the stack or the heap with malicious code [72], can also be exploited to manipulate automotive safety systems [78]. Modern processors can prevent arbitrary code execution through the OS kernel enforcement of the Writable xor eXecutable ($W \oplus X$) policy. Similarly, as shown in the previous chapter, AUTOSAR based systems can use the MPU to prevent RAM based execution of certain memory regions like the stack. With such protections in place, an attacker has to resort to Return Oriented Programming (ROP) attacks [89], in which gadgets are stitched together to execute code by jumping from one code snippet to another within the target system. To address the aforementioned types of attacks, several CFI schemes have been proposed over the years [24], [62], [46]. They typically instrument the code to insert additional checks for indirect branch and return instructions. The checks are made against a fixed call graph based on static analysis of the source code or binary. A special type of CFI techniques aims to reduce the explosion of branch checks by considering the context of a branch and return operation [99]. Context sensitive CFI(CCFI) promises more accurate branch restrictions based on checking a chain of edges upon reaching a critical program point. The idea of path monitoring is appealing for the automotive

use case because automotive systems usually have a limited set of functions that are the target of an attack. As seen in recently published attacks, normally the CAN transmission function is one of those targets that adversaries are trying to manipulate. CCFI is the base approach for the SecMonQ flow checker monitor. Applying CFI techniques in automotive systems faces several challenges. While several known CFI techniques have been shown to perform with an average CPU overhead of under 5% [37], deterministic systems are more concerned with the worst case performance rather than the average overhead. Moreover, CFI techniques that are fully implemented on the host would require code instrumentation of safety-critical code. This raises the cost of the CFI code as it now has to pass the rigorous requirements of safety-critical software development. This can be further exacerbated by continuous updates of the normal application code which would result in more validation of the re-instrumented code. SecMonQ aims to overcome these challenges by placing the bulk of the checks inside the HSM where the security monitor runs in an independent isolated environment. This results in a deterministic CPU overhead and the isolation of safety and security concerns.

5.3.2 Hardware Based Security Monitors

Wolf et al. [102], proposed a set of collaborating security monitors that are implemented in hardware to detect attacks on cyber-physical systems. The thermal monitor checks for anomalies in the device temperature, while a flow checker monitor detects anomalies in the control flow transitions. Specialized hardware is needed to interface the monitors to the host system and to take recovery action. Conceptually, SecMonQ follows a similar approach but without the need for specialized hardware. Instead, embedded HSMs are widely available in automotive micro-controllers which translates to no additional hardware cost. Also, our flow checker monitor is limited to checking for specific anomalies within safety and security execution paths. Var-

ious hardware based control flow integrity monitors have been proposed to reduce the impact of CPU overhead when performing flow integrity checks: [45],[23], [103], [67], [42]. [23] proposes an on-chip control flow monitoring module (OCFMM) which monitors the program counter and instruction register of the host CPU. It compares the control flow to a CFG stored in an isolated memory unit. OCFMM requires a generation tool to create the CFG and then load it into the isolated memory unit. While SecMon shares a similar high-level architecture as OCFMM, the use of a built-in HSM instead of specialized hardware, and the path generation approach make it more practical for automotive systems. In [103], an anomaly path detector is proposed with the help of a secure processor with special pipeline-checking hardware. Checking tables are created during a training phase and are stored in the host memory while being protected via the MMU against tampering by the host. Instead of checking a call graph, the execution path is broken into segments which are checked against the normal execution path. The approach requires specialized hardware as well as incorporating a training phase, which makes software updates costlier due to the need for retraining. [42] proposed a CFI enabled Instruction Set Architecture(ISA) with a hardware implemented shadow stack. The approach significantly reduces host CPU overhead, but at the cost of modified hardware architecture. Also, when a violation is detected, the host itself has to take action through an exception. The methods listed above have several benefits but are not quite suitable for the automotive use case. Also, the lack of safety focus clearly points us to the need for a specialized automotive solution to CFI defense.

5.4 Our Framework

For a security monitor to be effective, the system being monitored must exhibit a high level of determinism. This is to reduce falsely flagging normal events as anomalous. AUTOSAR based systems follow well-defined execution patterns which make

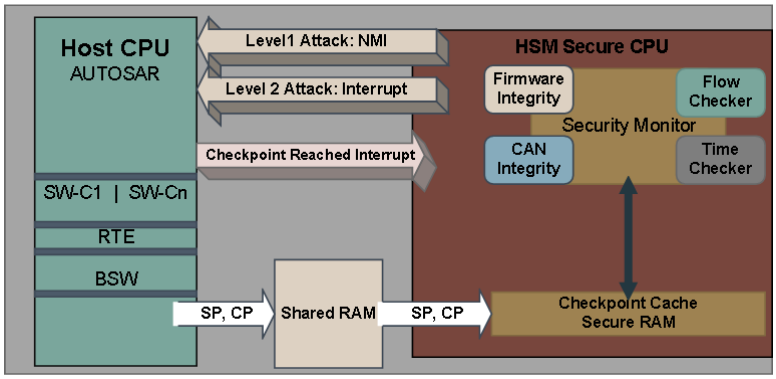

Figure 5.2: System Level Block Diagram

up the ground truth of the ECU. According to the AUTOSAR methodology, software application components and complex device drivers, shown in Figure 5.1, can only interact with the basic software(BSW) layers through the run time environment(RTE) layer. Due to the real-time nature of the ECU, Software Component(SW-C) runnables and basic software layers are triggered in a periodic fashion based on a statically defined schedule table. Moreover, communication peripherals are normally configured once at startup with a fixed set of parameters. For e.g. an ECU normally has a fixed set of CAN message ID's that it can send and receive. This high degree of determinism is one of the main motivations for pursuing a security monitor approach. In order to use the embedded HSM as a security monitor, the following requirements must be first met:

- The security monitor algorithms are executed in an isolated secure execution environment without direct host interference
- The security monitor is capable of inspecting host settings independently from the host
- A secure channel linking the host with the HSM is needed to report host states
- A secure interface linking the HSM and the host is needed to report attack events

- A trusted safety function within the host is needed to process the attack event data and transition the ECU into a fail-safe state

Next, we show how each security requirement is met with SecMonQ.

5.4.1 Secure Execution Environment

The HSM secure CPU shown in Figure 5.2, is based on the Bosch HSM architecture [36] making the solution portable to other automotive devices that support an embedded HSM. The BoschHSM architecture supports an independent CPU that executes its code from dedicated memory. It also contains exclusive memory for storing keys and other assets. SecMonQ runs as a software partition within the firmware of the HSM. This allows it to act as an independent monitor that is not directly influenced by the host. Moreover, an HSM typically has access to parts of the host memory and peripherals in order to perform monitoring independently [4]. This allows SecMonQ to perform monitoring activities without direct interference from the host.

5.4.2 Secure Checkpoint Buffer

The host CPU and the HSM communicate through a two-way interrupt mechanism coupled with a shared RAM buffer for message exchange, as shown in figure 5.2. Host execution events are represented with checkpoint identifiers. Entry and exit from a critical function translate into writing a checkpoint id into the shared RAM buffer. Typically, Trustzone [70] is not available in deeply embedded automotive MCU's, therefore, a malicious host can technically spoof checkpoint events by simply writing the checkpoint id into the shared buffer. Therefore, we protect the buffer via an AUTOSAR OS feature known as the trusted function approach [11]. AUTOSAR OS provides the ability to call a trusted function which belongs to a trusted application from a non-trusted application. Using the MPU, a dedicated shared RAM

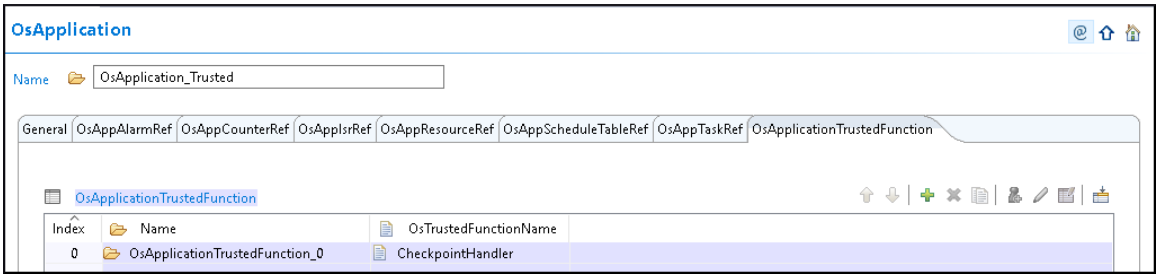

Figure 5.3: Configuring a trusted function within EB tresos

buffer can be configured as non-readable/write-able or executable in user mode. This makes direct access to the buffer only possible through the kernel and the trusted function. But simply restricting access to the buffer through the trusted function is not sufficient, because an attacker could call the trusted function to make it write a spoofed value. To cope with this, the trusted function performs a plausibility check of the reported checkpoint. This is possible through a hook inside the system call API in AUTOSAR OS which allows us to read the return address from the link pointer register and write it to the secure buffer. When the kernel switches into the trusted function, the buffered return address is cross-checked with the checkpoint id through an authenticated lookup table in general memory. The table itself is protected from tampering by SecMonQ which monitors the authenticity of the table on a periodic basis. This prevents an attacker from spoofing checkpoints.

5.4.3 Secure Interruption

When an attack is detected by the HSM, the mechanism of interrupting the host and forcing it back into a trusted state must be protected. Without protection, a malicious host can disable the interrupt mechanism to keep control of the host. In the hardware we evaluated, the interrupt mechanism from the HSM to the host is maskable. This is to allow the host to limit the number of interrupts that the HSM can generate which can interfere with the real-time performance of the host application. Therefore we propose two types of attack notification methods to the host. The

first uses the interrupt method which is subject to being disabled. This is used for anomalous events that are not considered severe. It is up to the system implementer to choose which events can use this notification method. For critical attacks that must result in host interruption to recover control, we make use of the non-maskable interrupt(NMI) feature on the host controller. The HSM has a physical port pin that shall be connected to the NMI input pin on the host CPU. When a severe attack is detected by SecMonQ, it asserts the NMI pin which forces the host to enter into the NMI exception handler. The host then executes code that safely shuts down the ECU and reboots the system into a trusted safe state.

5.4.4 Trusted Safety Function

When the host is interrupted, the function that controls the transitioning of the system to a fail-safe state must be trusted from a security point of view. The NMI exception code on the host is itself protected against modification because it is in flash memory that is periodically checked by the HSM for authenticity. The reason why we require that the transition to a fail-safe state be handled by the host rather than the HSM is that the latter does not have the full knowledge of the vehicle state. It is also not a safety qualified core and therefore it cannot take actions related directly to safety. By giving control back to the host, we ensure the principle of isolation is maintained where safety and security are separated but coordinated. From a safety point of view, we argue that an NMI is a proper response to a memory tampering attack because it is equivalent in severity to a non-recoverable memory corruption fault in the host. In that case, the host CPU issues a reset because he can not rely on the RAM data which has been corrupted.

In order for the security monitor to be truly effective, the following additional ECU security requirements must be applied:

- Only the RTOS shall be allowed to run in privileged mode. This may seem

obvious to security experts but AUTOSAR allows any safety-critical application to run in full privilege mode

- All stack memory shall be configured as non-executable via the MPU to prevent execution of injected code

5.5 Design of SecMonQ

As shown in Figure 5.2, SecMonQ consists of four monitoring components: *time checker*, *flow checker*, *firmware checker* and *CAN peripheral checker*. These monitors operate under two modes. The time and flow monitors rely on host application events which have to be reported by the host CPU to the HSM. While the CAN peripheral and firmware integrity monitors are executed autonomously by the HSM without direct host interference. For event reporting from the host, we initially leveraged the standard HSM job interface following the AUTOSAR crypto stack approach [6]. With the standard interface, the host creates a job object with unique job information then sends it to the HSM for processing. Instead of requesting a typical cryptographic job, the interface can be used to report an event such as a checkpoint being reached. But upon performing the time measurement of the communication overhead we determined that the job interface is not optimized for frequent event reports. Therefore, we implemented a lightweight interface that allows the host to set the checkpoint id and trigger an HSM interrupt without the need to wait for an acknowledgment. To prevent interrupts from being lost by the HSM, we added a check that the interrupt flag is cleared by the HSM before the host generates a new checkpoint event. On the HSM side, an interrupt service routine checks if the received interrupt is due to a checkpoint report and if so it buffers the event after adding a secure timestamp to mark the time it was received. The reason for caching these events is to reduce the time spent in the HSM interrupt service routine to allow the next checkpoint event

to be received with minimal delay. The actual sequence and time check is deferred to a periodic task that performs the monitoring activities inside the HSM firmware. That task loops through the received checkpoint events and performs the flow and time checks. On the other hand, the firmware and CAN integrity monitors perform their checks in a dedicated periodic task. In case any monitor detects a violation, it reports it immediately through the host interrupt or NMI assertion as described in section 5.4.3. Next, we examine each monitor design in detail.

5.5.1 Time Checker

The time monitor behaves like a security watchdog that detects missing or delayed safety tasks. The main purpose of this monitor is to detect attacks that reroute control to malicious code effectively disabling or delaying the execution of safety-critical tasks. But unlike a traditional watchdog which can be refreshed by an attacker who is executing code on the host, the time monitor cannot be easily spoofed due to the secure shared buffer mechanism described in section 5.4.2. Therefore missing the execution of periodic safety functions is guaranteed to raise a flag within SecMonQ. The time monitor takes as input the AUTOSAR OS generated schedule table [11], to determine the expected separation time between any two consecutive root nodes(tasks). A system expert chooses the tasks that must be monitored for timing during an initial setup phase. Examples of such checkpoints are application tasks that control vehicle safety functions, and basic software tasks that control vehicle communication. The resulting list is programmed securely into the HSM memory. During run-time, each monitored periodic task reports a checkpoint event upon its activation. If the time monitor does not receive any two consecutive timed checkpoints within a predefined time window then it flags the event as an attack and notifies the host.

5.5.2 Flow Checker

The SecMonQ flow checker is based on the context based control flow integrity(CCFI) approach presented in [99]. Given a security critical program point P and a path of control transfers $p = e_1, e_2, ..e_n$, the goal of CCFI is to check that $\forall i \in 1, 2, .., n$, edge e_i occurs in the correct sequence of preceding edges of the control flow graph(CFG). Instead of the program point being a security relevant kernel function, as is the case normally in typical computer systems, in our use case, P represents a safety or security critical edge function. In the context of AUTOSAR, examples of such functions are MCAL layer functions that directly interact with hardware actuation, sensing or communication. They could also be functions that control the security state of the ECU such as the diagnostic security access unlock function [7], which if entered maliciously would give an attacker access to perform privileged diagnostic services. During the setup phase, a system expert identifies the program points that must be protected against code reuse attacks based on the risk associated with such threats. For each chosen program point, a path of execution, referred to as secure path(SP), has to be defined and every function within that path shall be monitored as shown next.

5.5.2.1 Path Definition and Checkpoint Selection

The execution path of a program point can be generated with the help of a run-time profiling tool, which traces calls into the program point from various functions and generates a pruned control flow graph. For our target micro-controller, the Green-Hills profiling tool, [22], is able to perform this tracing. This one time process is needed to detect all functions that can lead to the critical program point. For each path that leads to our program point, we trace back until we reach the root node. The resulting path is given a unique SP id value. And each function on the path is given a checkpoint(CP) id value. Once all the functions on the critical path are iden-

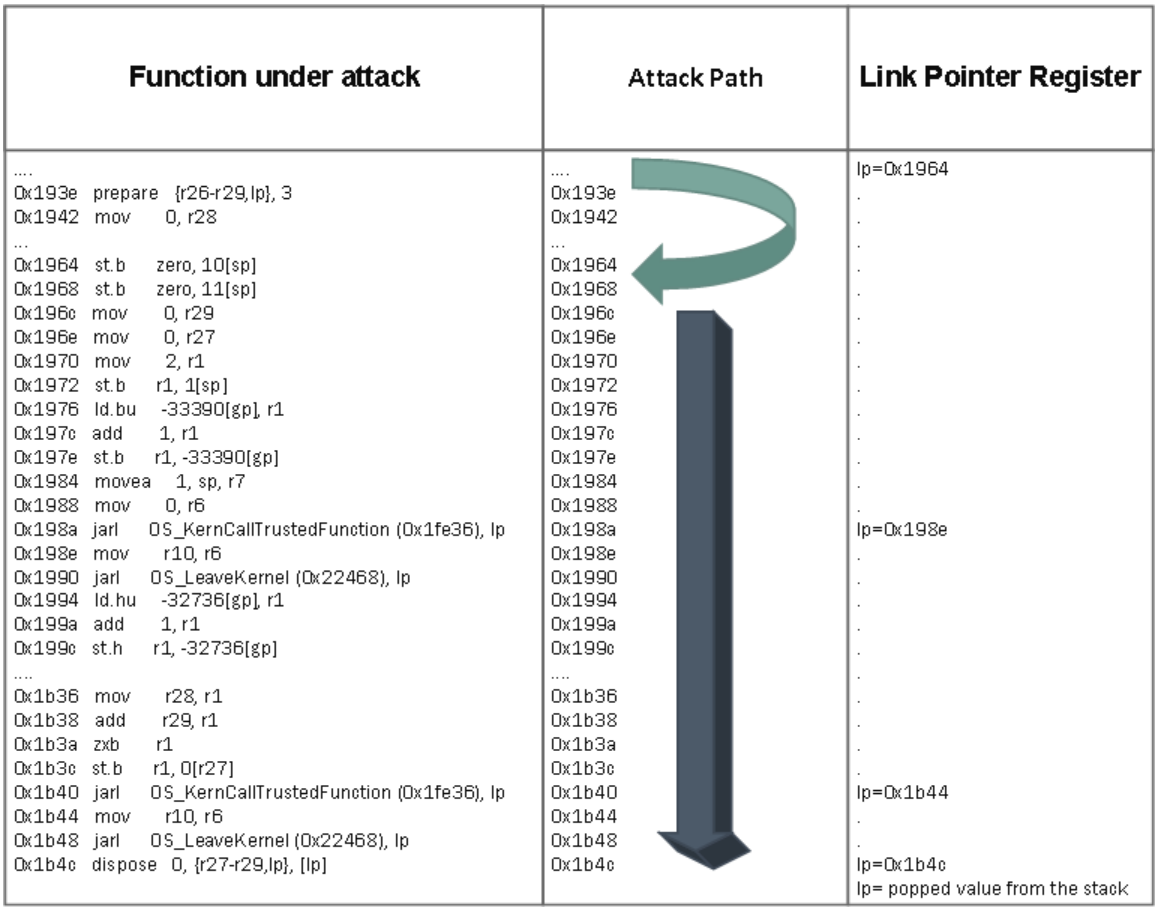

Figure 5.4: Attack flow against a function in the safety-critical path

tified, a forward scan is performed to identify exit functions that do not correspond to our monitored program point. The resulting additional branches must be included in our monitoring to ensure exits out of our secure path are possible without being flagged as an attack. An example pruned graph based on the AUTOSAR Can_Write function is shown in Figure 5.5. Furthermore, for each function on the SP, an entry and exit checkpoint is needed. To understand this point we examine the way functions are chained on our target CPU. The *jarl addr,lp* instruction stores the return address($pc + 4$), in the link pointer register: *lp*, then jumps to the address *addr*. Returning is done by executing a *jmp [lp]* instruction which retrieves the return address from *lp* and jumps back to the calling function. As seen in Figure 5.4, the function prologue pushes the return address in the link pointer register *lp* on the stack. To

return to the caller at the end of the function, the *lp* is popped from the stack and loaded into the *pc* register. An attacker can chain jumps by overwriting the stack with a sequence of addresses that skip the target function prologue. This ensures that at the end of each function, the address popped from the stack is the next address in the jump chain. Since the attacker can skip the prologue, he may also choose to skip the checkpoint entry, therefore, we insert an exit checkpoint to prevent complete evasion of our flow checker. For any given function, if the exit checkpoint is reached without the entry checkpoint being received, then SecMonQ flags an attack.

5.5.2.2 Flow Violation Detection

The flow checker periodic task fetches checkpoint tuples(SP_id, CP_id) from the checkpoint buffer in the order they were received. It then compares the received tuple to the preloaded graph of expected tuples for the active SP as shown in Algorithm 1. When a root node checkpoint is received, the flow checker sets the active SP to that root node. Since a checkpoint can belong to multiple SP's, the flow checker keeps track of the possible active SP's until the flow transitions into an SP with no overlapping branches. Due to the preemptive nature of AUTOSAR OS [11], it is possible that SP's that belong to tasks with different priorities preempt each other. Therefore, the detection algorithm allows switching from one active SP to another when a new root node checkpoint is received. For checkpoints corresponding to intersecting paths, the SP id is ignored by the flow checker. Instead of the checkpoint id following the sequence number, in this case it is assigned a unique label. This is necessary to allow the function to report a checkpoint regardless of which path it came from. Even though the SP id is ignored, transitioning into such functions is not possible from any arbitrary SP because the flow checker keeps track of the active SP and thus only allows transitions accordingly.

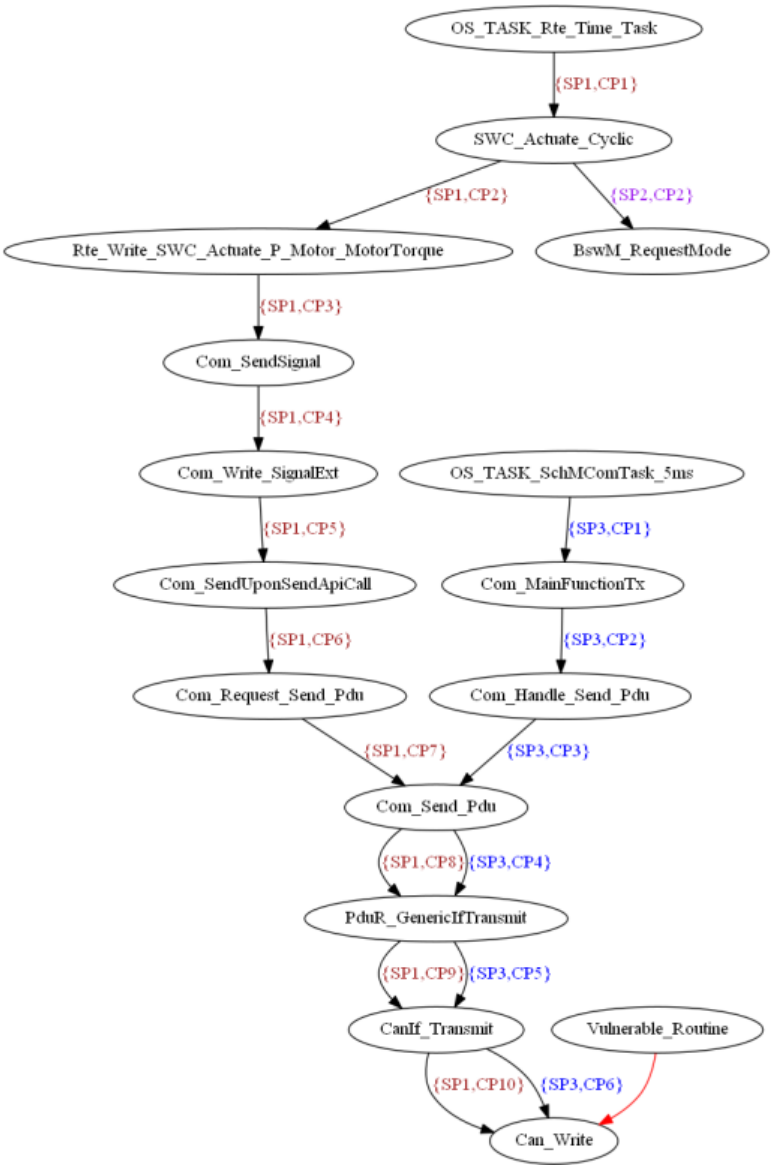

Figure 5.5: Control flow graph for the Can_Write critical function

5.5.3 Firmware Integrity Checker

The ability to modify the ECU firmware or calibration data has been the ultimate goal for anyone who wants to alter the vehicle behavior. The NIST security best practices mentioned in [25], include code signing to prevent attackers from altering vehicle firmware. Another security primitive that is essential to firmware integrity is secure boot [84]. While full boot measurement can be time-consuming at startup for automotive systems with strict availability requirements, background integrity

Algorithm 1 Attack Detection Algorithm

```
while CheckpointBuffer != empty do
    FetchFromQueue(SP(i),CP(j));
    if CP(j).type = TASK then // this is a root node
        // update active SP based on new received SP id
        SP.active = SP(i);
        // update expected checkpoint based on active SP call tree
        SP.expected → CP.expected = SP.active →CP.next
    else
        if SP(i) = SPx then
            // intersecting function, check unique CP label and ignore SP
            if CP(j) = SP.active → CP.expected then SP.expected → CP.expected
            = SP.active → CP.next ;
            else goto flagAttack ;
        else if SP(i) → CP(j) = SP.active → CP.expected then SP.expected →
         CP.expected = SP.active → CP.next ; // update CP to the next one in the
         SP sequence
        else // received checkpoint did not match expected checkpoint for the active
         SP
         goto flagAttack ;
    end
end
```

measurements performed inside the HSM present no timing penalty. The HSM can perform periodic boot measurements in the background when it is not busy with processing cryptographic jobs for the host. This is primarily the function of our firmware integrity monitor. During the firmware installation, the firmware monitor calculates an AES CMAC or a HASH of the downloaded firmware. The MAC or HASH value is stored in the HSM secure memory. During normal run-time, the integrity monitor performs an authenticity measurement using the built-in AES or HASH accelerators. If an attacker manages to bypass security checks and re-programs part of the firmware code or data, the firmware integrity monitor detects the manipulation and takes action within a predetermined time. The time to detect such tampering depends largely on the size of the data being authenticated. By partitioning the firmware code and data into blocks, it is possible to perform the integrity checks of critical code and data sections within a shorter period, before a full authentication measurement is finished.

An example of such a code block is the NMI exception handler which must be trusted to restore the ECU to a safe state. Therefore it is monitored with its own MAC tag which can be quickly verified without having to wait for the full firmware image to be authenticated to determine if it can remain trusted or not. If the NMI handler is no longer trusted due to an authentication failure, then the HSM locks its security assets and enters a reduced functionality mode to prevent an attacker from using it to authenticate transactions required for participating in secure vehicle operations. Here it is up to the system designer to decide on the recovery plan. At a minimum, an ECU that cannot pass a secure boot measurement is not allowed to participate in secure communication with the rest of the vehicle, and that is possible by the HSM restricting usage of its keys.

5.5.4 CAN Integrity Checker

The CAN masquerading attack requires the usage of a compromised ECU to send CAN messages that are not within its own CAN message transmit list. For e.g. a body controller that has been compromised, can be reprogrammed to send CAN messages that are received by the steering control ECU to manipulate the vehicle steering. Given the fact that the CAN controller settings are normally fixed after startup, and since the HSM has read access to the CAN registers, the CAN peripheral is a good candidate for security monitoring. The CAN integrity checker periodically monitors transmit control registers that contain the CAN identifier and data length code configuration of transmit messages. Additionally, the CAN integrity checker could monitor the CAN reception identifiers and mask filters to detect tampering with the reception rules. Such tampering would result in an ECU receiving messages that it would otherwise ignore, but we consider that out of scope. The Renesas CAN controller implementation in our device is called RSCANFD version 3 and shown in Figure 5.6. It supports 8 CAN channels, each with 32 transmit buffers and

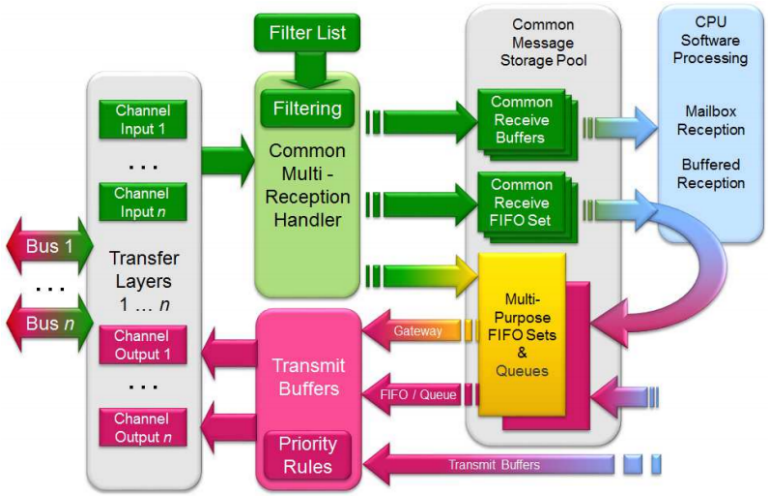

Figure 5.6: RSCANFD CAN controller architecture, [19]

operates in both CAN classic and CANFD [55] mode. Transmission can be done either with FIFO structures, prioritized queues or the traditional transmit mailboxes [19]. For simplicity, we only choose to monitor the traditional transmit mailboxes method. Protecting FIFO and the queue method will be considered in future work. Consequently, the registers that are mandatory for monitoring are:

- RCFDC0CFDTMIDp: sets the CAN frame type(classic vs. CANFD), the CAN ID value as well as the CAN frame ID type(standard vs. extended)
- RCFDC0CFDTMPTRp: sets the data length code value of the corresponding CAN frame

Note p is a value between 0 and 255 to cover all 8 CAN channels, thus we have to monitor a total of 512 registers. The remaining registers related to transmission are changed dynamically for e.g. to set the actual CAN data bytes or request transmission. Such registers are not suitable for monitoring with SecMonQ. The periodic rate at which SecMonQ must perform the CAN integrity checks must comply with the following inequation:

$$t_{checkerperiod} \leq n_{frames} * (t_{frame\_transmit} + t_{arbitration}) \tag{5.1}$$

a	d	dl	T_{canfd}
500 Kbps	2 Mbps	8	$127.2\mu s$
500 Kbps	2 Mbps	16	$165.6\mu s$
500 Kbps	2 Mbps	24	$204\mu s$
500 Kbps	2 Mbps	32	$242.4\mu s$
500 Kbps	2 Mbps	64	$396\mu s$

Table 5.1: Frame transmission times for various data lengths

where n_{frames} is the number of consecutive frames that the attacker must send back to back in order to override the real message on the bus and cause the intended harm, $t_{frame\_transmit}$ is the transmission time of each individual frame, and $t_{arbitration}$ is the arbitration delay before the CAN frame is transmitted. $t_{frame\_transmit}$ depends on the CAN bus baud-rate, message length and number of stuff bits resulting from the data payload. Using Equation 4.1, we calculate the transmission times for various frame lengths and commonly used baud-rate configurations, as shown in Table 5.1. In terms of the arbitration delay, it depends on the busload and the priority level of the message being transmitted. If an attacker is trying to send a message that has a low priority, then it can be delayed by several frame transmission times as it loses arbitration to the higher priority messages. On the other hand, a high priority frame will likely suffer one or no frame transmission delay. For safety messages, the data length is typically 8 bytes or longer, in order to accommodate the CAN signals, checksum or MAC, rolling counters and other parameters. In the worst case scenario where a CAN frame is an 8 bytes long high priority message, the CAN integrity monitor would have to check the CAN controller settings at a very high rate which places an impractical computational burden on the HSM. Therefore, we turn to a probabilistic approach that uses a practical $t_{checkerperiod}$ that can instead detect tampering in CAN register settings with a certain probability. Since the CAN integrity monitor is used in conjunction with the other monitors, we argue that this compromise is acceptable. Assuming that $T_{canfd} < t_{checkerperiod}$, then we can segment each $t_{checkerperiod}$ into equal slots of T_{canfd} time each. If the CAN integrity checker

reads the register values during the time segment where the malicious message is being transmitted, then detection occurs, otherwise, the attacker evades detection and simply rewrites the transmit ID with the original value to cover his tracks. Due to the nature of automotive control systems, the attacker cannot achieve his attack goal by sending a single malicious CAN frame. Typically, a control algorithm reads in data from the CAN bus continuously until the desired operation is completed. This forces the attacker to send the malicious message during that time to prevent the real ECU messages from being received, and consequently, aborting the malicious operation. To override the real ECU message, the attacker must send his messages with an attack rate, $t_{attackrate}$, faster than the real messages in order to either cause a collision or to overwrite the valid received data. Assuming an attack period, t_{attack}, in which the attacker is forced to send the malicious message m times, the probability of full evasion can be calculated by multiplying the probabilities of evasion during each instance that the CAN integrity checker is running, as follows:

$$P(\neg Det) = \left(1 - \frac{T_{canfd}}{t_{checkerperiod}}\right)^m \tag{5.2}$$

where $m = t_{attack}/t_{attackrate}$ and $P(\neg Det)$ is the probability of evasion, or simply put: $1 - P(Detection)$. Consequently, the probability of evasion decreases as the attack period and the number of required malicious frames increases. Note, if the attacker sends only a few malicious frames, i.e. below the threshold to override the real message, the probability of detection will be low, but the safety impact will be non-existent, as the real ECU will manage to send the valid messages causing the system to continue to operate safely. Moreover, to reduce the overhead on the HSM, we chose to perform the integrity check through an XOR operation between the register values and the expected values, instead of a MAC or hash calculation. If the XOR operation result is different from zero, then we flag a tampering event. Note, a MAC can be used

instead when the number of registers to be monitored is quite large which would place a significant storage overhead for the HSM secure memory. When the checker detects that the transmit buffer identifier values have been modified, SecMonQ locks access to the secure communication keys and notifies the host. Additionally, SecMonQ can leverage one of its IO port pins to assert the inhibit line on the CAN transceiver. This takes the CAN channel completely offline as the transceiver is no longer operating in normal state. To use the IO port pin option, the HSM must be physically wired to control the transceiver inhibit line, [18]. Incidentally, while the focus of the CAN integrity checker is to detect tampering with CAN transmission settings, the concept can be easily adapted for other communication controllers such as the LIN bus.

5.5.5 Policy Handler

The policy handler takes input from all the four monitors and then decides the proper action based on the pre-programmed policy actions. The HSM, being a master controller, can reset itself as well as the host cores. Our initial intuition was to allow the HSM to reset the system as a recovery mechanism to an attack. While this can be an acceptable response in typical IT systems, for safety-critical systems, such a reset can result in a sudden loss of control functionality which is, in fact, a safety goal violation. Since the HSM does not know what state the host is in, and is not equipped to take safety-related action, we decided to prohibit the HSM from resetting the entire system. Note, allocating safety-related action to the HSM would increase the automotive safety integrity level (ASIL) requirement of the HSM hardware and firmware adding cost and complexity to the system[17]. Instead, we designed the security monitor to delegate such action to the host safety core via the secure interrupt mechanism after taking actions from the action list in Table 5.2.

Nr.	Action
1	Set error state in a shared register with the host to inform of attack detection and reason
2	Dump out the host stack memory and write it to secure memory for future forensics
3	Lock the HSM security assets to prevent the host from successfully spoofing other ECU's with authenticated messages
4	Issue a maskable interrupt to the host to notify it of an attack that may require a transition into a fail-safe state
5	Issue a non-maskable interrupt to the host to force it to transition into a fail-safe state
6	Assert the IO pin that controls the CAN transceiver inhibit line to take the CAN channel completely offline

Table 5.2: Actions that SecMonQ can take upon policy violation

5.5.6 Safety Considerations

Due to the safety-critical nature of AUTOSAR based systems, several safety-related considerations are needed for SecMonQ. In terms of the peripheral monitoring, the host is expected to prohibit write access from the HSM into its own internal memory and register settings. This is in line with the freedom of interference concept that ISO26262 requires when elements with varying safety-criticality co-exist in a system. Read access, however, causes no safety violation and therefore, the HSM shall be permitted such access to the host peripherals. If a malicious host denies read access to the HSM, this would be considered an active attack and the HSM can take corrective action from the list outlined in Table 5.2. Another important safety factor is whether the attack detection time can be kept within the safety constraints of the system as shown in Figure 5.7. SecMonQ shall follow this rule for all its monitors:

$$t_{attackdetection} \leq FTTI - FRTI \tag{5.3}$$

where FRTI is the fault reaction time interval and FTTI is the fault tolerant time interval, as defined by ISO26262 [17].

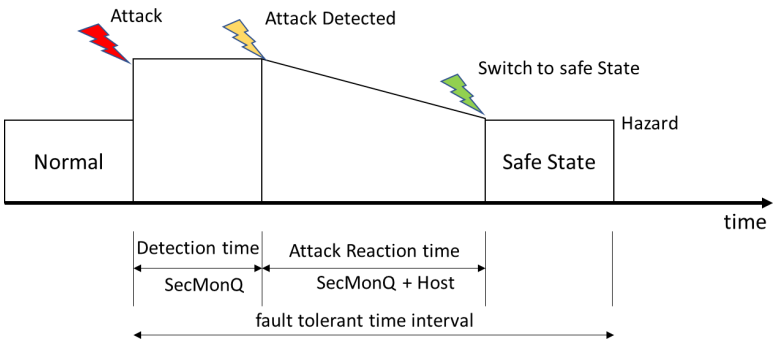

Figure 5.7: Impact of SecMonQ detection time on FTTI constraint

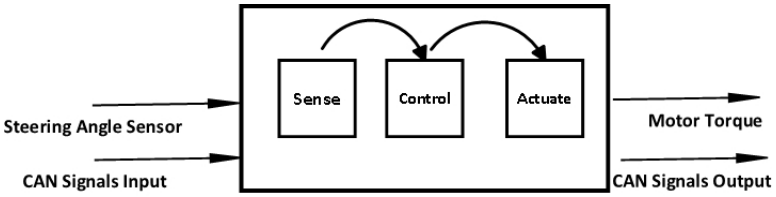

Figure 5.8: ECU Model of Sensor Actuator Application

Definition 1. According to Part 1 of the automotive functional safety standard [17], FTTI is the time span in which a fault or faults can be present in a system before a hazardous event occurs.

Figure 5.7 is adapted for security from the fault based model defined in the functional safety standard [17]. We make the argument that for embedded automotive systems, as long as the attack detection and reaction can be performed within the fault tolerant time interval(FTTI) [17], the system can still meet its safety objectives.

5.6 Case Study: Defeating the CAN Masquerading Attack

In prior work [78], we applied an early version of SecMonQ to the use case of protecting safety-critical routines which are triggered through the vehicle diagnostic commands. Protecting the diagnostic protocol[15], is a perfect candidate for SecMonQ because the ground truth can be easily modeled with a limited number of valid diagnostic command sequences that impact safety. In this chapter we consider the more complex use case of protecting an AUTOSAR based application from the

CAN masquerading attack, as described in section 5.5.4. Our test application consists of a single AUTOSAR software component(SW-C) that periodically reads in data from the CAN bus, increments a counter and sends out another CAN message again periodically. This is to model a simplified sensor actuator application as shown in Figure 5.8.

To evaluate SecMonQ, we construct several attack classes that aim to achieve the CAN masquerading attack objectives. In the first attack, the attacker attempts to modify the firmware to reprogram how CAN messages are transmitted. We expect that the firmware integrity monitor to detect that within the following time:

$$t_{detect} \leq t_{monitorperiod} + t_{authenticate} \tag{5.4}$$

where $t_{monitorperiod}$ is the task periodic time for executing the firmware integrity monitor in the HSM, and $t_{authenticate}$ is the time to calculate a MAC of the monitored code and data blocks and comparing it with a pre-stored authentication tag. The second attack involves RAM based code injection to redirect control to the Can_Write function in the MCAL layer. We apply the path generation steps to produce the SP,CP sequences corresponding to Can_Write function as shown in Figure 5.5. The attacker aims to reach the Can_Write function either directly by jumping into it, or through jumping into one of the functions that lead to it. To launch this attack, we overflow the stack frame of the vulnerable function with a payload of chained jumps that reach the Can_Write function. Consider $t_{Host2HSM}$ to be the time to switch to the trusted function and send the checkpoint event from the Host to the HSM, $t_{HSM2Host}$ to be the communication overhead from the HSM to the host to report the attack detection, $t_{HSMcheckerperiod}$ to be the periodic rate at which SecMonQ performs its monitoring checks, and $(t_n - t_{n-1})$ to be the time between any two consecutive checkpoint reports which result in a flow control violation, then the attack detection time must satisfy

the following rule:

$$t_{attackdetection} \leq t_{Host2HSM} + t_{HSMcheckerperiod} + (t_n - t_{n-1}) + t_{HSM2Host}. \quad (5.5)$$

The goal of SecMonQ is to minimize the detection time by minimizing the overhead for reporting a checkpoint as well reducing the rate at which the monitoring algorithms are executed without causing the HSM to be fully occupied with monitoring workload.

Next, we apply the rerouting control attack in which the attacker jumps into a function outside the monitored SP's and never returns back control. He can do that by jumping into an AUTOSAR critical section entry point which disables global interrupts or suspends the OS [11]. We expect the time monitor to detect missing checkpoints within a time window of:

$$t_{detect} \leq t_{monitorperiod} + t_{maxtaskcycle} \quad (5.6)$$

where $t_{maxtaskcycle}$ is the maximum separation time between consequtive activations of a periodic task. Finally, we inject a malicious routine that modifies the CAN controller registers related to the transmitted CAN message identifiers. We expect the CAN integrity monitor to detect this tampering before the vehicle function is successfully activated. Note, detection here is coupled with a probability value as described in Equation 6.3.

5.6.1 Experimental Setup: Time and Flow Monitors

Our experimental setup consists of the EB tresos AUTOSAR stack [2] version 4.0.3 with a single core configuration. The microcontroller used is a Renesas RH850 F1KM, shown in Figure 5.9, which is typically used in body control ECUs [21]. The latter is equipped with an HSM, also known as Intelligent Cryptographic Unit Master (ICU-M) [4]. In addition to being a security enabled micro-controller, it is

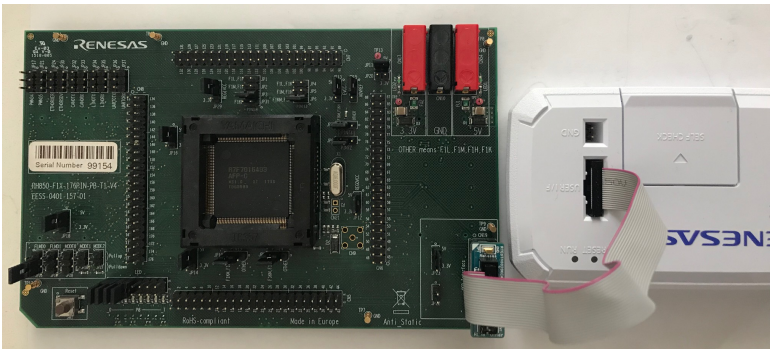

Figure 5.9: RH850 Development Environment and Debug Setup

also an ASIL qualified device. To interact with our ECU, we use the GreenHills Multi debugger which allows us to send commands over the debug interface as well as log data for further analysis. We configured the host core on the RH850 to run at a CPU clock frequency of 120 Mhz. For demonstration purposes we load the malicious stack memory payload used in our evaluation via the GreenHills debugger over the JTAG link. The overflowed stack contains a jump to the Can_Write routine located at a known flash address that the attacker determines by inspecting the firmware binary image. To observe the attack detection, we place one breakpoint in the Can_Write routine at the point where the transmission request is set in the hardware, and a second breakpoint in the NMI exception handler which is triggered by an attack detection. Then we corrupt the stack with our carefully crafted payload. We consider a missed attack as one that results in the Can_Write breakpoint being hit without the NMI detection breakpoint. We trigger the stack overflow at random points in the software execution flow to cause a jump into the Can_Write function via all the possible paths described in Figure 5.5. Note, even if Can_Write is entered successfully and a CAN message is sent, as long as the detection and recovery occurs within a fault tolerant time, the attacker can be prevented from doing real harm due to the built-in fault tolerance. In order to hook our checkpoint notification from the AUTOSAR application to the HSM, we take advantage of the AUTOSAR entry and exit hooks which are autogenerated in the RTE and BSW layers. The hooks can

be used for tracing certain events during development by mapping the macros to a trace tool. But we repurpose those macros to call the checkpoint handling function to report entry and exit from a function. The code in listing V.1 shows an example of tracing hooks for the Communication Layer function: Com_SendSignal.

Listing V.1: Com_SendSignal function containing the trace hooks at entry and exit of the function

```
FUNC(uint8, COM_CODE) Com_SendSignal
(
    Com_SignalIdType SignalId,
    P2CONST(void, AUTOMATIC, COM_APPL_DATA) SignalDataPtr
)
{
...
DBG_COM_SENDSIGNAL_ENTRY(SignalId,SignalDataPtr);
...

DBG_COM_SENDSIGNAL_EXIT(RetVal,SignalId,SignalDataPtr);
    return RetVal;
```

This allows us to integrate SecMonQ with AUTOSAR without the need to modify AUTOSAR specific interfaces.

5.6.1.1 Results

First, we targeted the function Com_SendSignal which belongs to the Com layer [16]. The function allows us to modify a CAN signal value which is later transmitted on the CAN bus by the CAN driver layer. The goal is to create a chain of two jumps, first to setup the modified CAN signal and second to jump directly to the Can_Write function to trigger the transmission. Using the memory map, we located

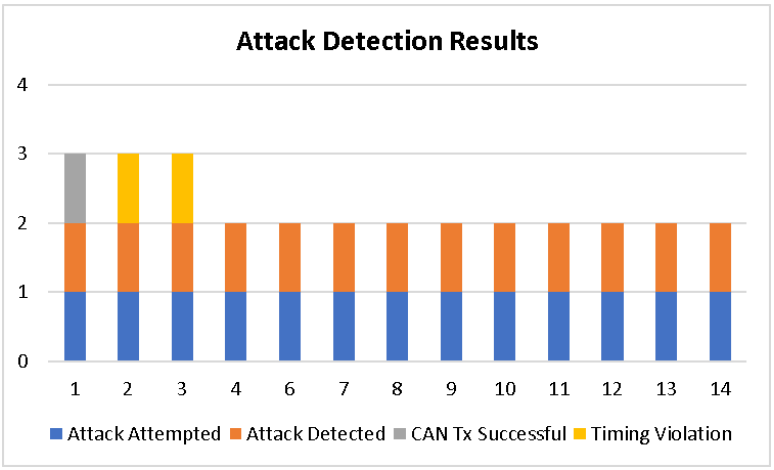

Figure 5.10: Attack results on the Can_Write function

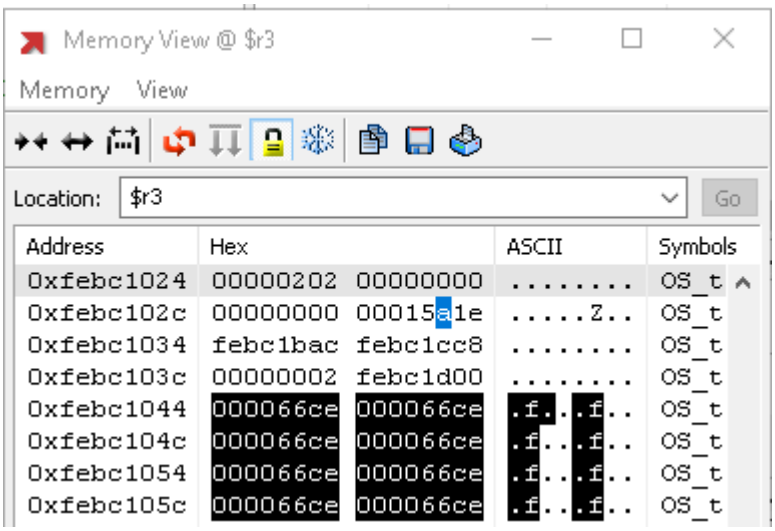

Figure 5.11: Contents of stack frame after corruption

Com_SendSignal at address 0x15a10 and Can_Write at 0x66ce. We then had to find the location on the stack where the *lp* register value must be corrupted. Through trial and error we determined the location of the *lp* and the rest of the registers which are popped from the stack of our assumed vulnerable function. Then we performed the code injection as shown in Figure 5.11. Instead of jumping to the start of Com_SendSignal, we skip the function prologue to avoid overwriting our next *lp* value by the prologue stack push. This way, once Com_SendSignal reaches the end, it again pops our next *lp* value and jumps to Can_Write. Since, our aim is to send our

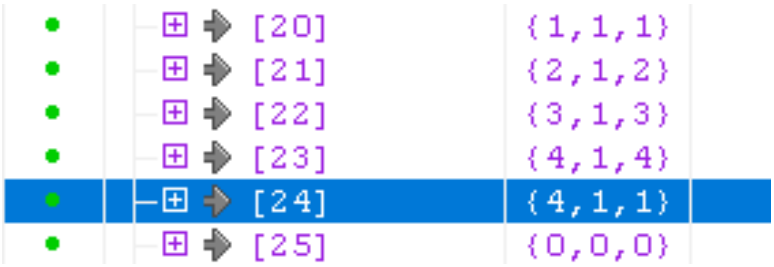

Figure 5.12: Checkpoint cache: reception of CP_4 when CP_1 was expected

spoofed CAN message continuously, we used the start address of Can_Write in the second jump. This results in the *lp* value being pushed on the stack at the start of Can_Write, and getting popped at the end, causing circular calls into Can_Write and effectively triggering the CAN transmission continuously. This is necessary to override the transmission of the actual message which is coming from the real ECU on the bus. Next we place our breakpoint in the Can_Write routine and the interrupt service routine which contains our attack detection code. By executing the attack, we can see that we entered the Can_Write function as expected, but the invalid checkpoint sequence triggers the attack interrupt service routine as shown in Figure 5.12. If the NMI is used, the sequence violation would have prevented reaching the Can_Write function on the entrance of the Can_Write function. But for convenience we relied on the attack detection interrupt to observe the behavior after each attack detection. One interesting finding was that after we launched the attack and observed the Can_Write function being continuously called, we verified that the OS was still able to schedule other tasks that needed to run as shown in the call trace in Figure 5.13. In effect, the task with the exploit was now converted into an endless loop of calling the Can_Write function, while other tasks could run as expected. This basically means that without the security monitor being present, an attacker could launch this attack without necessarily causing a crash thus continuing without detection. But due to our flow monitor, such attack gets detected and corrective action is taken. We repeated this attack scenario from various points against the secure path sequence and the detection

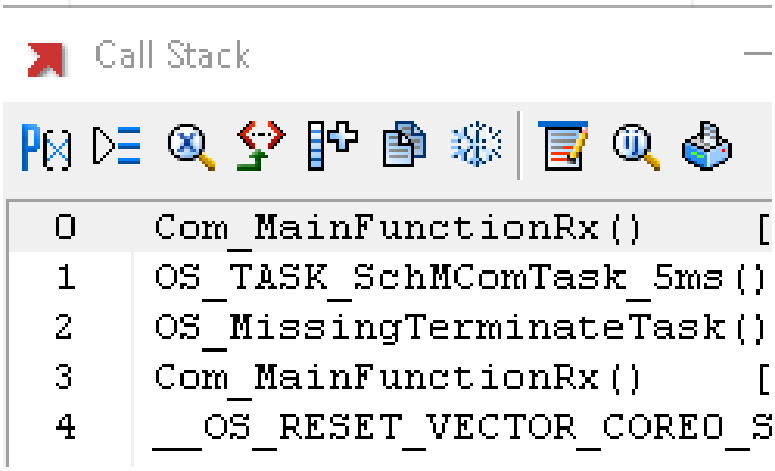

Figure 5.13: Call trace after the attack, OS still scheduling tasks

rate was 100% as seen in Figure 5.10. For the cases where we called the parent task to trigger the Can_Write function, a timing violation was detected in the time monitor. In one case when the attacker jumped directly into the Can_Write routine, the CAN transmit request was reached but detection occurred afterwards. Since SecMonQ issues an NMI, the attacker is unable to send any further CAN messages, thus causing no harm. In the remaining attacks, detection happened before the CAN transmission request could be set in hardware. In terms of checkpoint reporting overhead, we observed a significant difference, as shown in Table 5.3, when the checkpoint reporting is done through the AUTOSAR trusted function, vs. a normal function call. With the trusted function call AUTOSAR OS switches into the kernel mode which results in saving then restoring the current context. Since each checkpoint report adds a fixed delay, the overall delay within a single task depends on the length of the SP. In our experiment, the Can_Write SP required 22 checkpoints(including entry and exit), and this results in a total overhead of 2.32% during a single 10ms periodic cycle. We choose 10ms cycle rate because it is normally the common rate for AUTOSAR basic software components such as the CAN layers. It is important to note here that increasing the CPU clock frequency would have resulted in a lower overhead, but we chose to operate the chip at the slower clock frequency to show the worst case

	$t_{Host2HSM}(\mu sec)$	CPU Load/10ms cycle
Trusted Function	10.55	2.32%
Without Trusted Function	1.375	0.30%

Table 5.3: Checkpoint reporting overhead at host CPU clock of 120Mhz

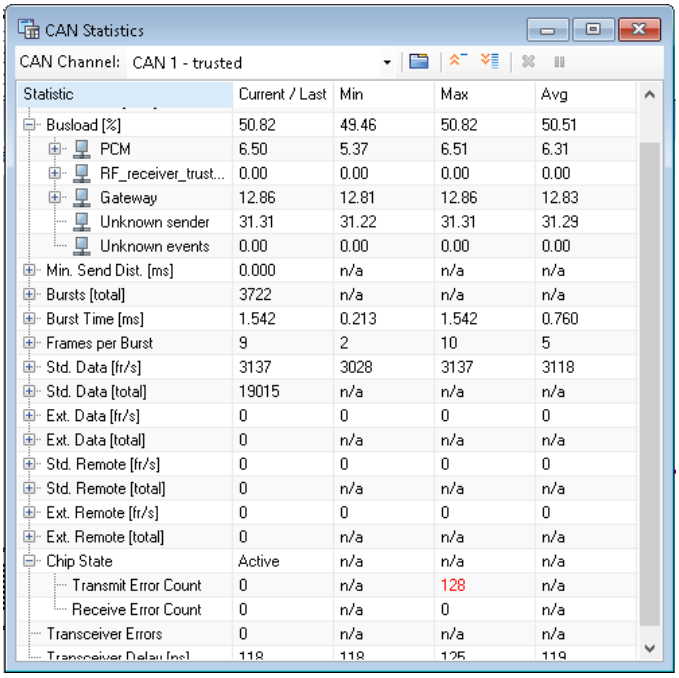

Figure 5.14: CAN Bus load to mimic real vehicle bus conditions

scenario. As for the $t_{HSM2Host}$, we measured 0.6μs with our HSM running at 60Mhz clock frequency. $t_{HSMtaskperiod}$ was set to 5ms to ensure the HSM has enough CPU bandwidth to handle normal cryptographic jobs. The attack reaction time, as shown in Figure 5.7, depends heavily on the implementation of the NMI exception handler and the amount of steps needed before the system can be switched to a safe state. In our case the combined detection and reaction time were observed to average around 8.6ms.

5.6.2 Experimental Setup: CAN Integrity Monitor

For the CAN integrity monitor testing, we use Vector CANalyzer and the CAPL language to simulate a malicious ECU and a driver assistance ECU which are communicating with a braking ECU. The malicious ECU aims at causing an emergency braking situation by spoofing the related CAN message. The CAN FD link was configured at 500kbps arbitration rate and 2Mbps data rate. The simulated driver assistance ECU sends a periodic message to the braking control ECU at a 20ms rate indicating no emergency braking needed. The simulated malicious ECU on the other hand sends the same message with the brake request signal enabled. Once the request is received, the braking ECU takes 250ms to start physically pumping brake fluid into the brakes to cause the deceleration. This time was given to us by discussing with domain experts. Therefore, the minimum attack time t_{attack} is 250ms. To ensure an attack does not result in a safety violation, the attack detection time must be less than t_{attack}. During that time, if any one of the driver assistance ECU messages is received successfully by the braking ECU, then the operation is aborted. The attacker thus has to continue to spoof the braking request message during that time to override the real messages. To simulate real vehicle conditions we create a CAN busload of around 50%, as shown in Figure 5.14, by scheduling several arbitrary CAN FD frames that are sent periodically to create bus traffic. Furthermore, we inject a jitter of 10% in the tranmission rate of the real braking request message. This mimics a real CAN bus where CAN messages do not get transmitted at exactly the same time due to loss of arbitration and varying busloads. Doing so is necessary to prevent attack messages from artificially lining up right after the real messages causing perfect overwrites.

5.6.2.1 Results

To determine the required attack rate that causes the successful attack, we vary the malicious message periodic rate starting at a value of 100ms down to 2ms. A

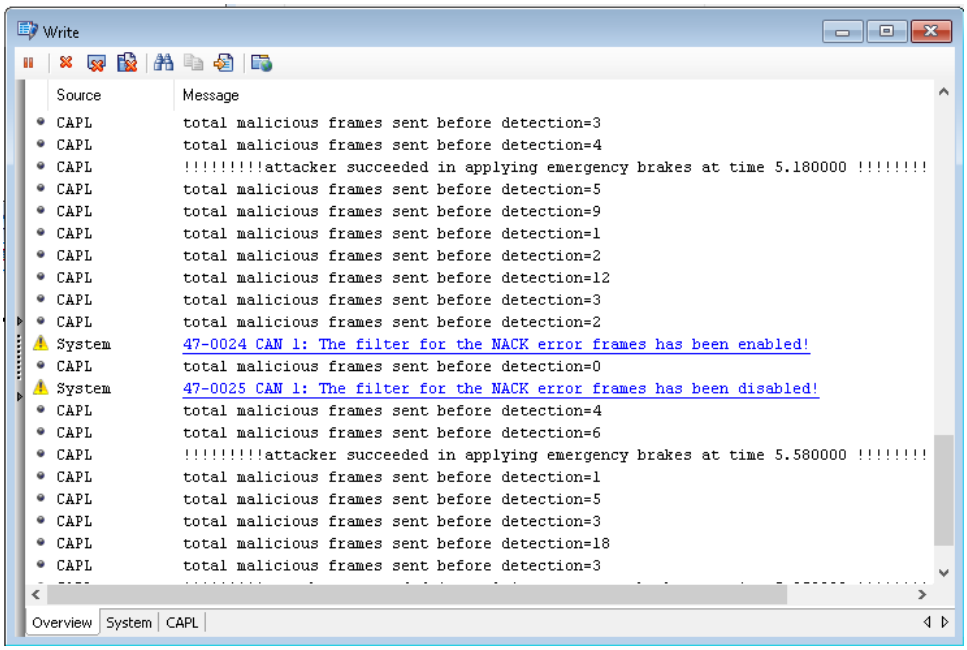

Figure 5.15: Attack results when attack rate is 10ms

CAPL routine that receives the CAN data at a 20ms rate, sets a flag that indicates the start of braking for a period of 250ms. During that time, if a real message is received, the routine aborts the operation. Otherwise, it prints a message indicating that the attack was successful, as shown in Figure 5.15. This indicates that the malicious brake request suppressed the real messages for the full attack duration of 250ms. As for the CAN integrity checker period, we chose a value of 1ms which is both computationally practical and also yields a high detection probability. Note, for testing purposes we did not enable the CAN integrity checker to take action when an attack is detected in order to allow the attacks to continue for data collection. As shown in Table 5.4, the attack is not successful until we reach half the periodic rate of the real message. Clearly, when the attack message is transmitted at twice the rate of the real message or faster, the attacker can overwrite the reception of the real message or cause a collision, causing the receiver to only see data from the malicious ECU. As we increase the rate even faster, successful attacks became more frequent. The experiment was repeated at a frame length of 64 bytes to study the impact of

Attack Rate	Successful Attack	Number of malicious frames	Prob(Det) (8byte frame)	Prob(Det) (64byte frame)
20ms+	no	less than 25	Don't Care	Don't Care
10ms	yes	25	96.4%	99.999%
5ms	yes	50	99.87%	100%
2ms	yes	125	99.999%	100%

Table 5.4: Attack results at different attack rates and Frame DLC

frame length on the detection rate. As expected, the detection rate improves as either the attack rate increases or the frame transmission time increases. Both increase the time window for the CAN integrity monitor to detect the register tampering. Note, for attack rates greater than the periodic rate of the real message, the detection rate dropped, however the attack itself did not result in a successful emergency brake operation. In terms of the CAN integrity monitor performance on the real hardware, we measured 96.6μsec to read all 512 register values and perform the XOR operation. The monitor executed this check once every 1ms.

5.7 Discussion

Embedded HSMs provide strong security guarantees in terms of isolation from the host. Since they are becoming standard in many popular automotive microcontrollers and SoC's, using them for security monitoring requires no additional hardware cost. However, there are some drawbacks that limit their scope of use as security monitors. With SecMonQ, knowledge about the host execution state, through checkpoints, requires switching into the kernel and back which results in considerable host CPU overhead for each reported event. Since our goal was to keep the implementation compatible with AUTOSAR, we chose the AUTOSAR trusted function which resulted in an approximately 8x increase in CPU overhead vs. sending the checkpoint report without the context switch, as shown in Table 5.3. It is possible to reduce that time

through an optimized method that utilizes the CPU software trap functionality directly to enter privileged mode. But, we did not pursue that route because we wanted to keep the solution compatible with AUTOSAR. The current approach results in a linear increase of the CPU overhead as the number of checkpoint reports increases. This limits the number of SP's that can be monitored without adversely impacting the real-time behavior of AUTOSAR. Future automotive microcontroller and SoC hardware architectures shall consider providing a hardware mechanism to reflect the host CPU state directly to the HSM. This would allow the HSM to observe execution events such as program counter values without the need for special software in the host. In terms of secure interruption, we had to utilize the NMI feature to ensure our attack detection interrupt cannot be ignored. But NMI is not a recoverable exception, therefore the host must shutdown itself and issue a reset at the end to recover the system. Having a hardware-supported security non-maskable interrupt would be desirable to allow the host to switch to a safe state without the need for a full shutdown at the end. Instead, the host could restart the offending OS application, which results in the increased overall availability of the ECU functionality.

Furthermore, one may argue that a ROP chain [89], can be used to execute arbitrary code without the need to enter the Can_Write function. However, ROP chains that rely on variable length assembly instructions gadgets are much slower compared to the normal execution flow. For e.g. the quicksort algorithm has been shown to take 100 times longer when entirely implemented as a return-oriented program [46]. This increased time works to the advantage of SecMonQ. If an attacker wants to mimic the functionality of the Can_Write function with ROP gadgets, he must perform register writes to the CAN controller. To avoid the normal AUTOSAR task from overwriting the modified CAN registers, the attacker must disable global interrupts as part of the attack chain. By doing so, he increases the detection window by both our CAN peripheral integrity monitor and the time monitor inside SecMonQ. This problem is

exacerbated by the fact that sending a single spoofed CAN message is not sufficient to cause real harm, so the process has to be repeated for a fixed number of times to ensure enough spoofed messages reach the target ECU, which further increases the chances of detection by the time and CAN integrity monitors.

When comparing with traditional host-based monitors, one advantage of SecMonQ is the clear isolation of safety and security which is especially important in safety-critical automotive systems. With host-based monitors, the safety-critical code must be instrumented with branch and return checks. But the resulting instrumented code is now the subject of safety validation. This results in significant cost to validate that the software is still safe after instrumentation. Similarly, there is a high cost and effort of safety qualification of the code instrumentation offline tools because they are generating code that is directly impacting the safe execution [17]. With SecMonQ, only the checkpoint report mechanism in the host and the NMI handler must be shown to be safe. Since the bulk of the logic is performed inside the HSM, there is no fluctuating CPU overhead that can happen in the host. Additionally, the HSM code that controls the NMI must be developed with safety in mind to prevent spurious NMI events to the host which can result in continuous shutdowns. Still, this translates in significant cost reduction due to the lower safety burden placed on the HSM software as a whole.

5.8 Conclusion

In this chapter, we presented a security monitoring system that addresses several security gaps which were uncovered in the previous chapter. The four components of SecMonQ allow us to detect CAN controller tampering, OS timing violations, flow control violations, and application code tampering. To demonstrate effectiveness we applied the solution to the popular CAN masquerading attack. It is easy to see that the approach can be extended to other use cases like protecting restricted

diagnostic services and flash programming routines. Due to the increased availability of embedded HSMs in automotive ECUs, SecMonQ adds no hardware cost and is compatible with AUTOSAR which results in minimal impact on legacy software. SecMonQ also defines a harmonized approach to attack detection within a safety environment by delineating fail-safe action to the host, while the HSM performs the security protections of its assets. This harmonized approach between safety and security shows how safety-critical systems can separate responsibilities within a single system yet harmonize the reaction when a cyber attack with safety impact is detected. By shifting the bulk of the monitoring to the HSM, we ensure freedom of interference between the safety-related host software and the HSM security monitoring software. This leads to lower cost for safety validation of the overall system. Furthermore, the approach can be implemented in any HSM equipped ECU. For future work, we plan on adding new monitors for clock and power glitch detection by utilizing the HSM built-in clock monitoring system. The CAN integrity monitor can also be extended to detect changes in the receive filters configuration. We also aim to investigate the impact of monitoring when multi-core systems are employed.

CHAPTER VI

Availability: Adding Trust without Sacrificing Performance

6.1 Introduction

In the previous chapter, we saw that the firmware integrity monitor was a critical component for ensuring that firmware manipulations in flash are detected and action is taken to bring the system back to a safe state. For the firmware integrity monitor to be effective in a safety-critical system, manipulations of flash have to be detected within a constrained timing rule. But the firmware integrity checking time increases as the size of the image in flash increases. This calls for a solution that accelerates firmware authentication to meet the strict timing constraints of automotive systems. In this chapter, we introduce a novel approach to accelerate the firmware authentication time based on our work in [77]. The approach has two purposes: reducing the time to detect a flash tampering attack during runtime, and meeting the startup timing requirements for real-time embedded systems without foregoing secure boot.

Dependable embedded systems like automotive Electronic Control Units (ECUs) are expected to perform an integrity measurement on startup, through either a Cyclic Redundancy Check (CRC) or a checksum calculation, to prevent corrupted software from being executed [83]. With the introduction of cyber threats against such sys-

tems, it is not sufficient to only look for memory corruption errors, but also to perform a data authenticity check to ensure the software has not been tampered with while at rest. This is a prerequisite for allowing the software to perform safety related functions. For automotive systems, a secure boot measurement is typically performed with the help of hardware accelerators in Trusted Platform Modules (TPMs), or embedded Hardware Security Modules (HSMs) [36], [44]. Although cryptographic hardware accelerators improve the boot authentication time, the ever increasing demand for memory makes it challenging to perform a boot measurement of the entire software while still meeting the availability timing requirements of such systems. The emergence of ADAS and autonomous driving systems further increases the memory requirements resulting in even longer times to perform a full boot measurement. For real-time systems that run within the context of an RTOS, partitioning the application into independent components that can be started in stages is often impractical especially with the need to support legacy software. Even with a staged boot approach, the full system functionality cannot be available until all the stages have been completed, causing overall delays in the startup time.

To address the security and availability requirement of real-time embedded systems, this chapter presents a dual phase secure boot approach that leverages an embedded hardware trust anchor such as an HSM to perform a sampled boot authentication step followed by a full boot authentication step. The probabilistic boot authentication sampling scheme(PBASS), uses pseudo random functions(PRFs) and message authentication codes(MACs) to randomly sample the firmware space and authenticate it. The general scheme relies on a true random seed that is generated during the setup phase and stored in secure memory. Using a PRF, the seed produces a set of memory blocks that are selected for the MAC calculation. During the verification phase, the seed is fetched from secure memory and the PRF is used to generate the sampled memory blocks to perform the MAC calculation. The calculated MAC

is compared to the stored MAC and if a mismatch is detected then the firmware is considered tampered. It is assumed that a hardware trust anchor is available to store the security parameters in a protected environment. PBASS shall fulfill the following requirements:

- Achieve high detection probability of data tampering
- Provide significant startup time improvement when compared to traditional secure boot approaches
- Resist attackers who attempt to evade detection by guessing the unsampled blocks
- Support firmware update strategies

6.2 Related Work

Secure boot is the process by which a hardware trust anchor evaluates the integrity of the software running on a device to ensure the device is booted in a secure state [69]. It is considered a fundamental security primitive of trusted computing [84], [40],[88]. Secure elements play a major role in building trust in a system [90], [52], [97]. In the case of a TPM[27], a well defined process of hash measurements is performed at each boot stage to be compared to a signed hash. The boot process continues if the calculated hash matches the expected hash until the full software has been authenticated successfully. Failure at any stage can cause the boot process to halt or cause a security reset. In the case of authenticated boot, the boot process is allowed to continue but the integrity values will be stored to be later reported for e.g. when connecting to a cloud server that requires remote attestation. Alternatively to TPM's, EVITA [87], proposes an embedded hardware security module [44], to support the main automotive security use cases. EVITA offers three profiles: light,

medium and full. The medium and full profiles, offer a secure execution environment that allows supporting various secure boot strategies. Another popular security architecture in automotive devices is the BoschHSM[36], introduced in Chapter V. The secure boot approach presented in this chapter is well-suited for an embedded HSM due to the flexibility of such platforms. The commonly known secure boot approaches in automotive systems can be summarized as follows:

- Full Secure Boot: here the full binary image is verified in a single operation. It is suitable for devices that execute from internal flash and have to authenticate a relatively small application. The advantage of this approach is that there is no need for partitioning the firmware into independent blocks. The downside is that as the application grows, the time to finish the authentication can easily exceed the startup time requirements.

- Staged Boot: with this approach the application is partitioned into multiple images that have to be booted sequentially, prioritizing partitions that are time critical over less time critical software partitions. A hardware trust anchor is normally used to securely boot an initial partition which is then trusted to boot the next stage. The drawback of this approach in embedded systems that run from internal flash, is that it requires segmenting the software into several partitions that can be executed independently. For deeply embedded devices that run a single binary image, this is hard to achieve. Also, the full functionality of the system is not possible until all partitions have been authenticated. In the case of bigger devices that have heterogeneous cores and boot from external flash, staged boot allows various cores to be booted in parallel which results in an overall improvement in startup time.

- Just in Time: here the system verifies executables before they are booted into RAM. This can improve timing but when various executables depend on each

other, the overhead of constantly booting code can become prohibitive. It is also only usable for SoC's in which software is executed exclusively from RAM.

Regardless of which secure element or boot strategy is used, the issue of competing goals between dependability and security remains. This was well-studied in [56], where the use of TPM's in Cyber Physical systems was shown to impact various aspects of dependable systems. One such aspect is the delay of availability due to the computational overhead of cryptographic protocols [86]. This is the problem we aim to solve in this chapter.

Surveying the literature, we did not find any work related to probabilistic secure boot schemes. However in the work of [28], a probabilistic authentication scheme was proposed for proof of possession of data between a client and a cloud server. In that use case, the client gives the cloud his files and is concerned the server is lying about possession of all his files. As a countermeasure, he challenges the cloud on a periodic basis by requesting it to calculate authentication tags of pre-sampled data blocks. Since the server does not know which blocks are covered by the authentication tags he cannot delete large data blocks without risking detection by the client. While the use case is different than secure boot, the underlying principal is similar to what we are proposing here. Instead of a malicious server that has the incentive to modify large number of blocks of user data without detection, we have an attacker who wants to tamper with a device software to change its behavior. But unlike the cloud use case, our attacker does not necessarily modify a large number of blocks making it harder to detect tampering. Therefore, we extend the probabilistic authentication scheme to cope with small block modifications and apply it to embedded systems showing that it can be practical for accelerating the secure boot process.

6.3 PBASS: Probabilistic Boot Authentication Sampling Scheme

In this section we describe the different variants of PBASS which were designed to deal with the security requirements listed in the introduction. Each variant aims to solve deficiencies of the previous variant until we reach the final variant that satisfies all our requirements.

6.3.1 Threat Model

We assume a powerful adversary who has managed to find a way to reflash the firmware on the device. The adversary also knows the details about the secure boot algorithm and how sampling is done, so he aims to modify blocks of memory that are outside the sampled space in order to evade detection. Due to the storage of the random seed, keys and MACs in secure memory, and the difficulty of inverting the PRF to recover the seed, the attacker can only guess which parts of the firmware are outside the sampling area through trial and error. We assume the adversary has direct access to the device and can repeat the process of modifying the firmware as many times as he wants until he can create an image that can evade detection of the sampled boot phase.

6.3.2 Notation

- d – total data blocks in the firmware image.
- $m \in d : 1 \leq m \leq d$ – modified data blocks.
- B – block length in bytes.
- $q = m \times B$ – modified data bytes.
- $s \in d : 1 \leq s \leq d$ – *randomly* sampled data blocks.
- $b : 1 \leq b \leq B$ – *randomly* sampled data bytes per block.

- $T : 1 \leq T \leq \frac{d}{B}$ – modified data blocks with $t > 0$.
- $t \in T : 1 \leq t \leq B$ – modified bytes per block.
- $P(Det) = P(m \cap s)$ – probability of detection of a modification to d.
- $P(\neg Det) = 1 - P(Det)$ – probability of detection evasion.

6.3.3 Random Block Sampling:rBS

Following the general scheme of Section 6.1, this variant also consists of a setup phase and a verification phase. The setup phase is depicted in Figure 6.1, while the details are shown in Algorithm 2. During the setup phase a random seed is generated along with the corresponding pseudo random numbers list. The random values are translated to memory block addresses to be used in the sampled boot measurement. The conversion of random bits into random block addresses consists of selecting a number of consecutive random bits and then applying a mask with an offset to produce a valid memory address. The mask is needed to limit the range of values in the random sequence and align it to an addressable block. The offset is needed to convert the random sequence into a valid memory address in the firmware image address map. A random key is also generated to be used for the tag generation/verification of each sampled image. The corresponding boot tag is calculated and stored in secure memory along with the corresponding key. During startup, the embedded HSM performs the verification boot measurement using the stored random seed and corresponding key to verify the authenticity of the firmware sample as shown in Algorithm 3. The sequence is identical to the setup phase with the exception that the random seed is fetched from memory instead of being generated and the MAC is compared to the stored value. Due to the size of the sample being a fraction of the full image, the secure boot time measurement is also a fraction of the total secure boot time. Note that the block size is chosen based on the optimal fetching speed of the hardware and the

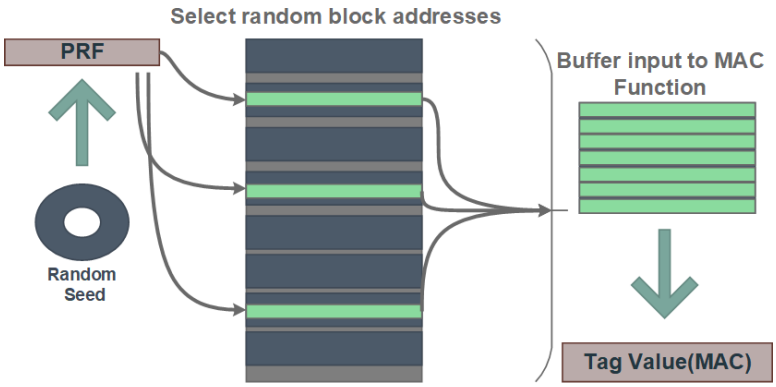

Figure 6.1: Tag generation process using randomly selected blocks

underlying MAC algorithm. For AES CMAC, the minimum block size shall be 16 bytes to avoid wasting clock cycles in packing the data into 16 byte aligned blocks. For other MAC algorithms, the block size can be adjusted accordingly to optimize performance. Moreover, PBASS can only be practical if it can detect tampering of data blocks with a high probability. Our sampling method is equivalent to a sampling with replacement approach, since the number of blocks remains the same after each sampling event. Therefore, the probability of detection after a single sampling event i is $P(Det_i) = \frac{m}{d}$, where m is the number of modified blocks and d is the total number of blocks in the firmware image. Consider M to be the set of modified blocks while S is the set of sampled blocks, then the adversary evades detection if $M \cap S = \phi$. Consider $P(\neg Det_i)$ as the probability of missed detection for each sample i, then

$$P(\neg Det_i) = (1 - P(Det_i)) = \left(1 - \frac{m}{d}\right) \tag{6.1}$$

Since all the samples i are *random* they are also independent events, therefore, to evade detection, the attacker has to escape each one of the sampling events, i.e:

$$P(\neg Det_s) = P(\neg Det_1) \times P(\neg Det_2) \times \ldots P(\neg Det_s) \tag{6.2}$$

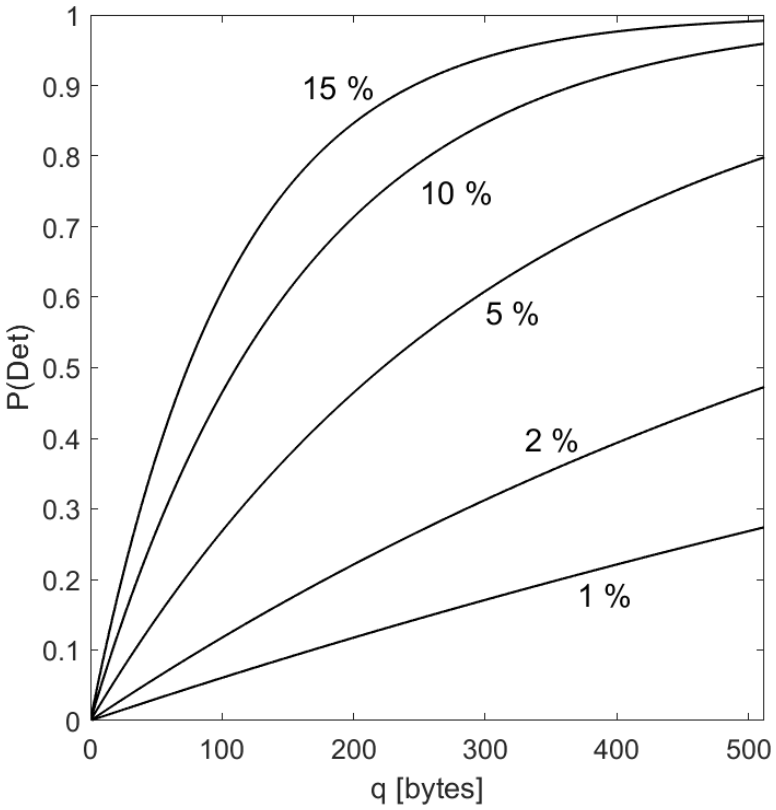

Figure 6.2: Theoretical probability of detection vs. number of *contiguously* modified bytes given various sample sizes using rBS

Consequently:

$$P(\neg Det_s) = P(\neg Det_i)^s = \left(1 - \frac{m}{d}\right)^s \tag{6.3}$$

For a given maximum value of m modified blocks and a maximum probability of evasion, $P(\neg Det)$, both of which the system can tolerate, we can represent the required sample size as follows:

$$s = \frac{\ln\left(P(\neg Det)\right)}{\ln\left(1 - \frac{m}{d}\right)} \tag{6.4}$$

Figure 6.2 shows the $P(Det)$ for various sample levels and different values of m. Note m was converted to bytes by multiplying it with a block size of 16. Intuitively, the larger the sample size is the higher the probability of detection should be. Moreover, the smaller the number of modified bytes is, the harder it should be to detect the attacker's modification. Interestingly, at a sample size of 15%, the detection

probability converges to 1 as long as m is sufficiently large. To see the impact of the image size on the evasion probability, Table 6.1 shows the calculated probabilities given a sample size of 15% and a fixed number of modified bytes of 512 bytes. Another notable point with this scheme, is that the attacker is motivated to pack his

Image Size(KB)	$P(\neg Det)$
128	0.81528%
256	0.81912%
512	0.82105%
1024	0.82201%
2048	0.82249%
4096	0.82273%

Table 6.1: Impact of file size on evasion, s=15% and m = 512 bytes

modification in full blocks to increase his chances of evasion. In other words, if the attacker distributes the modified bytes in an un-contiguous way, then he effectively modifies more m blocks than he would if he modified bytes in complete blocks. To keep the speed performance high, we need to minimize the sampling rate, but doing so limits the number of modified blocks that we can detect with rBS. This leads us to the next scheme.

6.3.4 Per Block Sampling:pBS

The previous variant showed high detection probability but only if the number of tampered bytes were large enough. This might not be suitable for applications of high criticality in which even a small program modification can render the system unsafe. To cope with low values of m, we extend PBASS to perform a double sampling approach. The first level of sampling is done at the full firmware image level by dividing it into smaller blocks. The second level of sampling is done at each one of these smaller blocks by sampling b number of bytes as shown in Figure 6.3. In this scheme, each block is guaranteed to be sampled b number of times. Assuming that the attacker modifies a minimum of t bytes over a single block of size B, and that we

Algorithm 2 Sampled Boot Setup

Input: Choose parameters: $K, scheme, n$
Output: $S_{stored}[1:n], T_{stored}[1:n]$
Function `generateTag`$(S, K, scheme)$**:**
    $R_{bytes}[] \leftarrow$ `PRF`(S)
    switch *scheme* **do**
        case rBS **do**
            for $i \leftarrow 1$ **to** s **do**
                $R_{addrs}[i] \leftarrow$ `derMemAdd`$(R_{bytes}[])$
                $R_{data}[1:B] =$ `getDataBlock`$(R_{addrs}[i])$
                $T_i \leftarrow$ `updateMAC`$(R_{data}[], K)$
            end
        end
        case pBS **do**
            for $i \leftarrow 1$ **to** d **do**
                $R_{addr}[1:b] \leftarrow$ `derMemAdd`$(R_{bytes}[])$
                $R_{data}[1:b] =$ `getDataByte`$(R_{addr}[1:b])$
                if $len(R_{data}) = MAC\_len$ **then**
                    $T_i \leftarrow$ `updateMAC`$(R_{data}[], K)$
                else
                    Continue
                end
            end
        end
    end
return T
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow TRNG$
    $S_{stored}[i] \leftarrow S$
    $K_{stored}[i] \leftarrow K$
    $T_{stored}[i] \leftarrow$ `generateTag`$(S, K, scheme)$
end

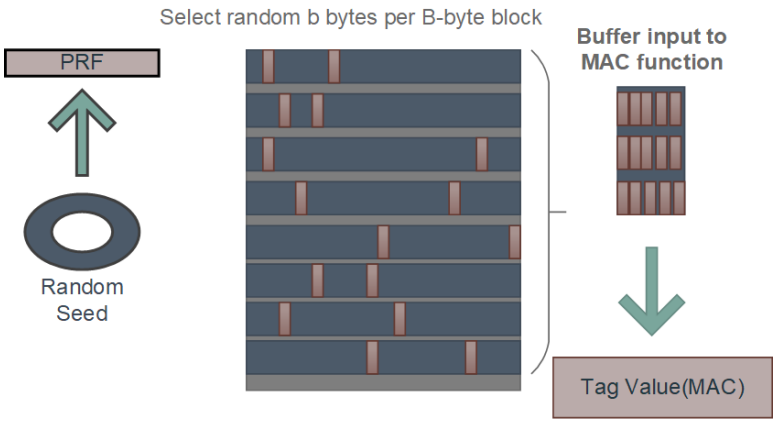

Figure 6.3: Per Block Sampling method

are sampling b bytes out of every block in the firmware image, then the probability of evasion can be calculated as follows:

$$P(\neg Det_i) = (1 - P(Det_i)) = \left(1 - \frac{t}{B}\right)^b \tag{6.5}$$

But since the attacker cannot fit his entire program into a single block, he has to spread the malicious code over T blocks. Thus, to evade detection he has to do so in each one of the T blocks. Following the same reasoning of Equation 6.2 we arrive at this equation:

$$P(\neg Det_s) = P(\neg Det_i)^T = (1 - \frac{t}{B})^{b*T} \tag{6.6}$$

Intuitively, by sampling each block in the image, the detection sensitivity rises significantly as T rises. This is evident in Figure 6.4, where $B = 16$ bytes while $b = 1$ and $b = 2$. We observe that the detection probability quickly converges to 1 even for small values of T modified blocks, each with a single modified byte, i.e. $t = 1$. When sampling individual bytes per block, the byte fetching and buffer packing operations slow down the overall performance in comparison to variant 1 which fetches complete blocks. Therefore, to keep the performance fast, the sampling ratio b/B has to remain relatively small. For a given total number of modified bytes that the system can tolerate $T * t$, an acceptable evasion probability $P(\neg Det)$, and a

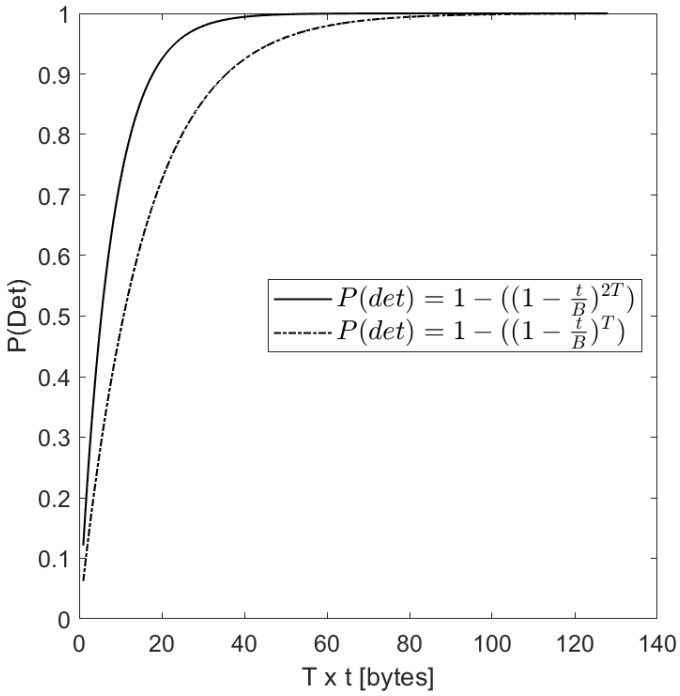

Figure 6.4: Theoretical probability of detection vs. number of modified bytes for per Block Sampling (pBS), with B=16 bytes, t=1 byte and b either 1 or 2 bytes per block

block size granularity B, we can calculate the required b value as follows:

$$b = \frac{\ln\left(P(\neg Det)\right)}{T * \ln\left(1 - \frac{t}{B}\right)} \tag{6.7}$$

Note here that the firmware image size is not factored in this equation. The reason is that independently of the image size, we are sampling each block in the image. Therefore, the sampling rate is simply b/B. It is important to note here that unlike the previous variant, the attacker can no longer pack his data in full blocks without risking immediate detection. The reason is that since each block is being sampled, if he modifies all values in just a single block, he will cause 100% detection probability. This forces the attacker to distribute his code modifications over as many blocks as he can, i.e. t has to be as small as possible. But to allow the malicious code to operate continuously, the attacker has to insert jump instructions to chain the modified blocks. This increases the amount of modified bytes needed to realize the

attack goal. Depending on the CPU architecture, a minimum t value can be fixed to account for the minimum instruction length plus the jump instruction length.

6.3.5 Resisting Adversary Evasion

Both schemes presented so far can be subverted by an attacker who aims to avoid detection by only modifying blocks which are outside the sampled region. Note here, we use the term blocks to refer to either multi-byte blocks or single byte blocks. Due to the secret nature of the random seed, to discover the unsampled blocks, the attacker modifies blocks one at a time and cycles power to rerun the boot verification and determine if the block is within the sampled space. Due to the flash architecture, it is not possible to just program a single byte, so the attacker has to erase the entire sector containing the target block, then reprogram all blocks in that sector with their original values except for the modified block. If the system is degraded, he infers that his modification was detected and so he marks the block as sampled and moves on to the next block. This process is repeated until the attacker maps out enough blocks that are out of the sampled space and then creates the program modification that only changes the unsampled blocks. To activate his malicious program, he needs to hook into one of the untampered software routines. To defeat this attack, we extend both variants 1 and 2 in two ways. First, we add a second phase boot measurement that is performed by the HSM in the background after the system is operational in order to fully authenticate the firmware. If the firmware is verified successfully, the HSM re-executes the setup phase to generate a new random seed, a new MAC key and a new MAC of the resulting sample space. If the full boot measurement fails, the HSM switches off the sampled boot measurement until the system has at least been fully booted successfully once. This frustrates the attacker who now needs to replace his modified blocks with the valid image to allow full boot to pass, before he can repeat his experiments of searching for unsampled blocks. However, since full boot

generated a completely new random seed, the attacker loses all the knowledge gained from previous evasions. This in essence limits the attacker to guess unsampled blocks from a single cycle. As long as the selected probability of detection is sufficiently high, the attacker cannot rely on guessing the unsampled blocks from a single measurement run. To further frustrate the attacker, we extend the setup phase to generate n random seeds. This produces n random samples which PBASS will now select from randomly at startup. This means, the attacker not only has to discover unsampled blocks in a single measurement run, but also in all the sampled measurements. That is because the attacker cannot guess which seed will be selected during startup. Note, that having n sampled spaces does not degrade the boot measurement time at startup because only one single space is verified. However, it does increase the setup time which now has to be multiplied by n setup instances.

6.3.6 Security Properties

PBASS has several security properties that make it hard for attackers to bypass:

- Although the firmware image is the same for ECUs of the same type, the measured sample is different due to the different random seeds. So even if an attacker is successful in tampering with one ECU, he cannot scale the unsampled space knowledge to other ECUs.

- Even if the attacker finds some bytes that he can modify without detection, he is limited severely to the time window that the ECU is active before he has to cycle power to evade detection by the second phase boot authentication. In case an attacker is cycling power to prevent a full boot measurement to be reached, the HSM can count power resets which would trigger the switch from sampled boot to full secure boot on each subsequent cycle. This would cause immediate detection of tampering and allow the ECU to take appropriate action. A sliding window approach can be applied where if m out n attempts fail, then we assume

that tampering is ongoing. As a result we switch to a single phase full boot authentication approach until a successful secure boot measurement has been performed. Then, we switch back to the dual phase approach.

Algorithm 3 Sampled Boot Verification

Input: $S_{stored}[1:n], T_{stored}[1:n], K_{stored}[1:n], scheme$
Output: True/False
$n \leftarrow rnd(1:n)$
$S \leftarrow S_{stored}[n]$
$K \leftarrow K_{stored}[n]$
$T \leftarrow$ `generateTag`$(S, K, scheme)$
if $T = T_{stored}[n]$ **then**
    Run Host
    Initiate Full Boot Authentication
else
    $Sample\_Boot\_Allowed\_Flag \leftarrow False$
end

6.3.7 Support for Firmware Reprogramming

Firmware in embedded devices is expected to be updated at some point in the lifetime of the device. PBASS easily integrates with firmware reprogramming strategies through the firmware verification step. After the firmware is downloaded into the device, it is expected that a verification step is performed to validate the digital signature of the firmware. It is after this point that the setup phase of PBASS is executed to generate the set of random seeds, keys and MAC's, to be used for the subsequent boot cycle. By incorporating the setup phase during the update process, we ensure that no startup burden is placed on the device. A secure programming event is equivalent to a full boot measurement, therefore on the next boot up event, the device software is allowed to be authenticated using the sampled boot approach.

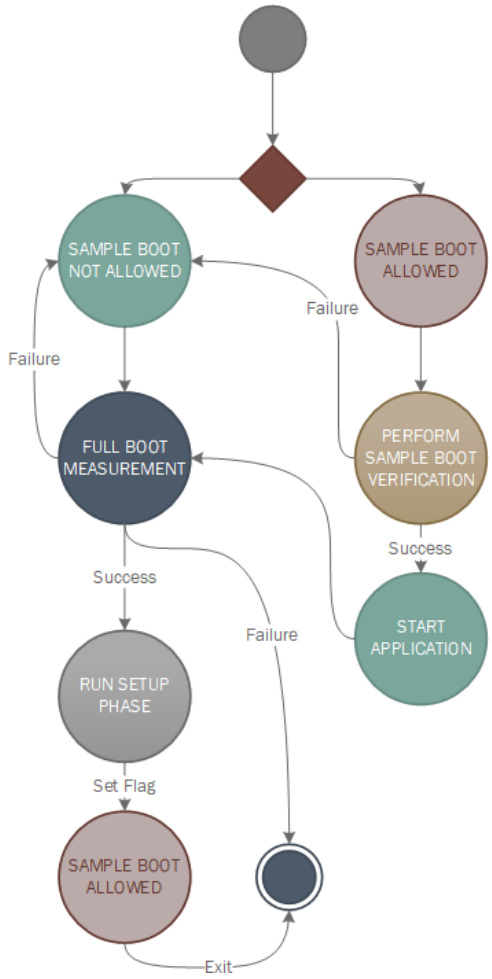

Figure 6.5: State flow diagram for the dual phase boot approach

6.3.8 PBASS: The Complete Approach

Now that we have shown how PBASS can meet the security requirements set forth in the introduction, we present the complete dual phase boot flow in Figure 6.5. For devices that require a single secure boot authentication stage, such as ones that execute out of internal flash, the PBASS flow replaces that single stage. For devices that require multiple secure boot authentication stages, such as ones that boot from external flash, the PBASS flow can be selected to run within individual stages where acceleration is desirable. This makes PBASS compatible with existing secure boot strategies. Assuming the firmware has not yet been securely programmed, then the sampled boot verification is not allowed. The system has to either perform at least

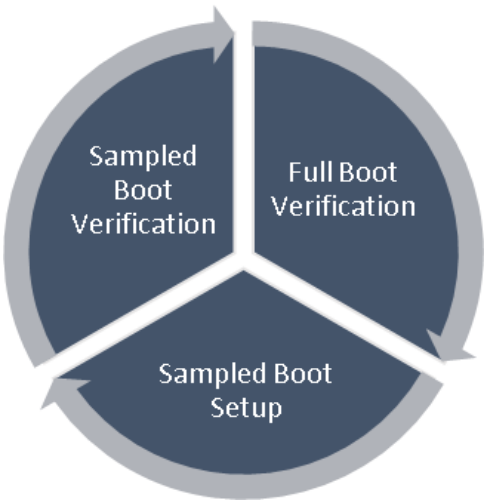

Figure 6.6: Coupling of the sampled boot phase with the full boot phase to reduce risk of undetected data tampering

a single full boot measurement following a failed boot authentication attempt, or undergo a secure programming event in order to transition into the sampled boot algorithm. The next boot cycle, the "sample boot allowed" flag is checked and if TRUE, then the flow for sample boot verification is executed. After a successful verification, the application is started and a full boot measurement is triggered in the background. A successful full boot measurement leads to a new setup phase initiation to generate the new random seed set and corresponding tags. This is illustrated in Figure 6.6. Moreover, if during any boot cycle, the sample boot verification fails, the "sample boot flag" is cleared forcing the device to switch back to the full boot measurement approach.

6.4 Implementation and Results

To evaluate our scheme we started first with a python simulation. Since the detection rate is hardware independent, we setup a python script that uses the open source pycrypto toolkit[68], to perform the underlying cryptographic functions. For the random block sampling(rBS) simulation, we follow the algorithm to generate a

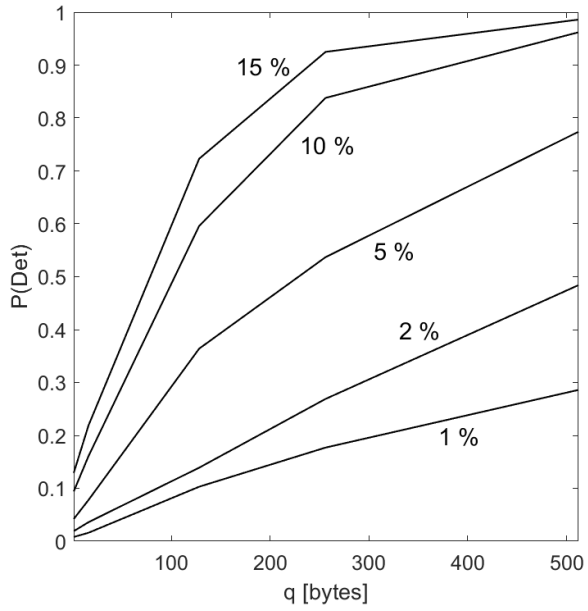

Figure 6.7: Python Simulation showing probability of detection vs. number of *contiguously* modified bytes given various sample sizes for Random Block Sampling (rBS)

list of sample addresses within our memory address range. For the modified block addresses, first we use the urandom python library to generate a list of attack addresses that correspond to q *contiguous* bytes, and make up m *contiguous* block modifications. As discussed earlier, packing the modified blocks is in the interest of the attacker so we simulate that scenario first. To test whether rBS detects the modification, we check if for any address in the attack list, the address also exists in the sampled list. The assumption here is that if the attacker modifies a single byte within the sampled block set, the MAC value will no longer match leading to a detection of tampering. For each set of the tests, we choose a sample size s from the following list: 1%, 2%,5%,10%, 15% and a q value from the following list: 1, 16, 128, 256, 512 bytes. For statistical significance the test for each single combination of q and s is repeated 1000 times, with the list of attack addresses changing each time. The results are shown in Figure 6.7. Comparing with Figure 6.3, we can see that the simulation agrees with the theoretical model with the probability of detection approaching 1 for $q \geq 512$ bytes and $s \geq 15\%$ of d. To demonstrate the need for the attacker to pack his modifications, the attack

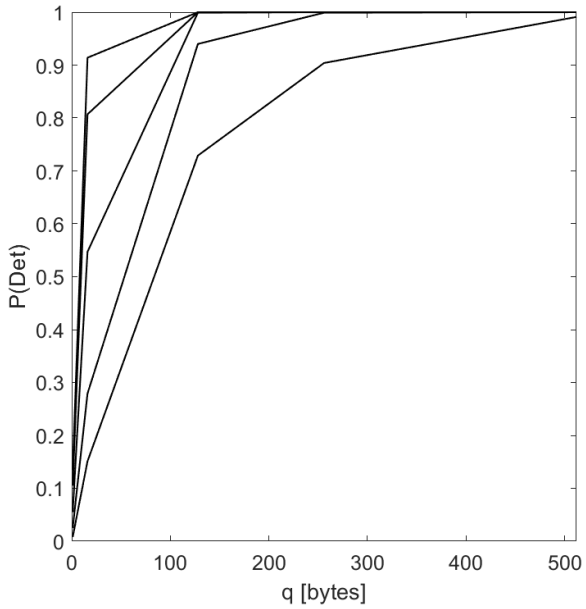

Figure 6.8: Python Simulation showing probability of detection vs. number of *non-contiguously* modified bytes given various sample sizes for Random Block Sampling (rBS)

addresses for the second simulation consist of *randomly* modified bytes. As seen in Figure 6.8, spreading the modified bytes greatly reduces the attacker's chances for evasion. Next we turn to the per block sampling method (pBS). Two simulations are run, one with $b = 1$ i.e. 1 sampled address per block, and the other with $b = 2$. Since modifying any single block entirely now results in a definite detection, with pBS the attacker now has an incentive to spread out his modifications. Therefore, we generate a list of attack addresses, which are completely random irrespective of block location. Since the attacker wants to maximize the distribution of bytes to evade detection, we choose $t = 1$ which is the worst case scenario. Even though this is not practical for an attacker who wants to chain instructions through jumps, it allows us to evaluate our system's performance even in the case of an attacker doing the illogical thing of just corrupting data. As with rBS, we repeat each test 1000 times. Note: Instead of a single q value, the modified bytes now have two components: T and t. The results are shown in Figure 6.9. Comparing with Figure 6.4, we can see that the simulation agrees with the theoretical model with the probability of detection approaching 1 for

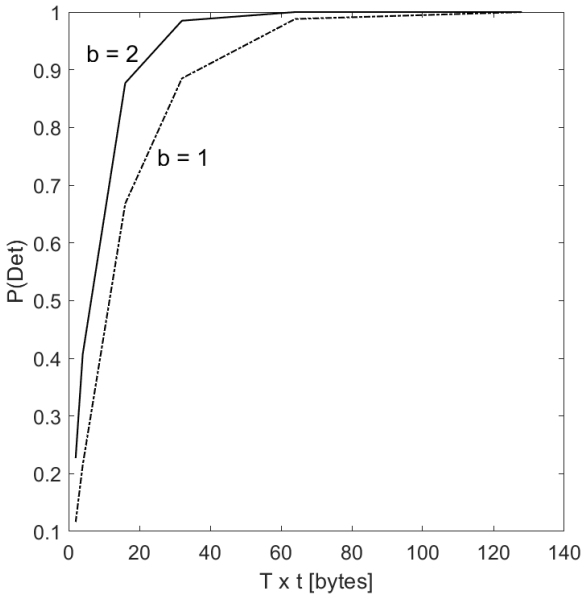

Figure 6.9: Python Simulation:probability of detection vs. modified bytes for different b values using pBS

$T \times t \geq 80$ bytes when $b = 1$, and $T \times t \geq 40$ bytes when $b = 2$.

To measure the timing performance we switch to the real hardware. One of the assumptions for implementing PBASS in an embedded system is that a secure execution environment with secure storage is available. This is mandatory to be able to store the random seed, keys and MAC values of the sampled blocks. Also, the full boot phase has to be executed using a trusted entity and the HSM is ideal for this due to it's secure core that runs an independent firmware. Therefore, we chose a popular automotive microcontroller:Renesas RH850 which has an embedded HSM[4]. We implemented an HSM firmware service that takes as input the boot measurement phase(setup or verify) and returns a job result. The HSM firmware performed the random seed generation using a built-in true random generator. The PRF was implemented as the NIST recommended AES CTR DRBG function [33]. The MAC was implemented using AES CMAC[94]. The choice for a CMAC was driven mainly by the fact that all automotive MCU's equipped with EVITA-MEDIUM or EVITA-FULL HSM have AES hardware accelerators. The algorithm can easily be adapted to

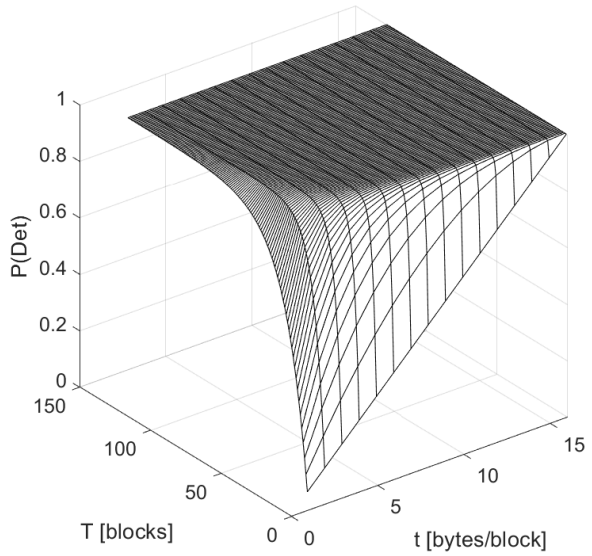

Figure 6.10: Probability of detection vs. number of modified bytes per block(t), vs. number of tampered blocks(T) for per Block Sampling (pBS) with b=1.

work with other hardware accelerators such as a SHA engine. Although the sampling approach reduces the size of data that has to be processed with the AES engine, the additional overhead to generate the pseudo random numbers as well as fetching blocks or bytes from memory to construct the AES data payload can be time consuming. For that reason, we optimized the algorithm to generate a large set of random numbers which are buffered in HSM RAM as a first step. Then we constructed our sample addresses using the random numbers buffer and fed the data into the AES engine through multi-segment requests of 16 bytes each. Due to the RAM size limitation, the process of generating random bytes may have to be repeated several times depending on the total sample size. We adapted the C version of the algorithm into python and made it available at [76]. In terms of memory overhead, each sampled space requires the storage of a 32 byte random seed, a 16-byte AES CMAC tag and a 16-byte AES key. If n seeds are used, the total storage needed is multiplied by n. A single flag to indicate whether sampled boot is allowed or not must be stored in the HSM secure memory. The actual implementation of the sampled boot algorithm required less than 500 bytes of flash memory. As for the RAM consumption, since the

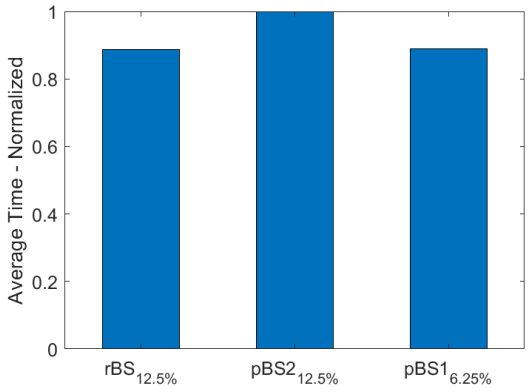

Figure 6.11: Setup time comparison between the different variants with different sample sizes

sampled boot verification step is performed only during startup, any RAM buffers used by this algorithm can be freed after the verification is complete. This makes the RAM usage impact minimal. For running the tests, we used the GreenHills MULTI environment which allowed us to send UART based commands to trigger the boot authentication jobs. First we issued a setup request to generate the random seeds and MACs. Then we issued the verification request to verify a preloaded firmware image. Subsequently, prior to each verification job request, we manipulated a fixed number of bytes with a specific pattern:"0xA5". We then tabulated the result of the verification phase to compare with the simulation without finding a significant difference. With each modification, we performed ten different sample verifications based on ten different seeds and measured the time. This was done to speed up the testing. We also measured the time taken to perform a full boot measurement of the entire image for comparison. We chose $P(\neg Det) = 0.0005$ and $m = 128$ bytes as representative values of the risk level acceptable along with the tolerance for undetected tampering. Figure 6.11 shows the setup time performance when comparing the random block sampling approach at a 12.5% sampling rate vs. the per block sampling approach at two sampling rates of 12.5% and 6.25%. As expected, to achieve a similar setup time to rBS we had to sacrifice the sampling rate in pBS to counter the inefficiency of fetching individual bytes from memory rather than fetching entire blocks.

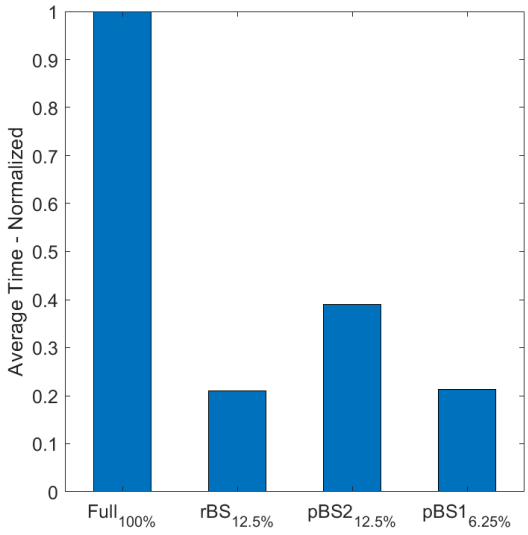

Figure 6.12: Verification time comparison relative to full boot

Figure 6.12 shows the verification time performance as compared to the full boot approach. The pBS algorithm at a sampling rate of 6.25% performed almost as good as the rBS algorithm at 12.5% sampling rate. Both of which performed at about 22% of the full boot time, approximately a 5X speed improvement.

6.5 Discussion

Our experiments showed a 5X speed improvement at 99.95% detection probability with a tolerance of up to 128 modified bytes. Due to the second full boot measurement, the risk tolerance is only for the time window from startup until the full boot measurement has finished. By delaying the full boot measurement until the system is operational, we are suggesting a two-level transitive trust approach. A successful full boot authentication operation infers a degree of trust that allows the second boot cycle to rely on the sampled boot phase, followed again by a full boot authentication phase that re-affirms trust in the environment and acts as an anti-tampering countermeasure.

PBASS comes with a few limitations that must be considered. While an attacker is restricted by how many bytes he can modify to perform something useful in code,

modifying data is not so restrictive. Take for e.g. calibration data or safety-critical constant data. An attacker may be interested in modifying only a few bytes to achieve his aim of chip tuning or causing a safety hazard. For data that is considered safety or security related we propose that a separate authentication scheme is applied to prevent undetected tampering. Instead of performing an aggregated verification at startup of all data, the application would verify the authenticity of such data elements on-demand. This would reduce the impact on boot up time while ensuring that data is not consumed if it has not been authenticated first. Another limitation is related to the boot failure strategy in case the background full boot authentication operation detects tampered memory. By switching to full boot mode, this means the ECU will now take longer to startup on the next ignition cycle because trust is degraded in the environment. But this is not a severe restriction because if tampering is detected, the ECU would have to be degraded anyway until someone restores it to a valid state through reprogramming. And since the setup phase is repeated as part of the next reprogramming cycle(needed to recover the ECU to a valid state), if programmed successfully, the following ignition cycle PBASS will run the sampled boot phase so the ECU experiences no added delay.

With pBS, the attacker is forced to spread his malicious modification over as many blocks as possible. If he wants to jump from one code modification area to another, then he has to add jump instructions to each code snippet. This can force him to use high t values and, depending on the complexity of his malicious code modification, a large number of T blocks. Alternatively, the attacker can modify individual bytes by carefully constructing instructions in a ROP attack style[89]. Instead of modifying return addresses through the stack or the heap, the attacker would modify blocks containing jump addresses in order to chain various software gadgets. But ROP based gadget execution is known to be slow compared to normal execution because it requires many gadgets to achieve what normal code would. This results in a high

number of modified bytes which again causes an increased probability of detection. Moreover, the attacker is not freely able to use any gadget because at each startup we select a different sampled space forcing the attacker to generate many gadget combinations that he has to try, which makes such attack impractical. In terms of performance, When a Direct Memory Access Controller(DMAC) is available in the hardware, it can happen that full boot measurement is close in time to a sampled boot measurement due to the increased speed of data fetching into the hardware accelerators. Without the speed performance edge, PBASS would become unfavorable when compared to a full boot authentication. We argue that DMAC can also make PBASS faster. In the case of rBS, DMAC would allow us to use larger s values with greater block sizes. Instead of sampling 16-byte blocks at a time, PBASS would sample multiple blocks to take advantage of the increased speed of data fetching. Since the overall data sample would still be a fraction of what full boot measurement has to process, we anticipate that PBASS would still be superior. Evaluating with DMAC will be a target for a future extension of this work.

6.6 Conclusion

In this chapter we presented a probabilistic boot sampling authentication scheme that provides a high degree of confidence in the integrity of the booted software. This allows resource constrained embedded systems to speed up their boot authentication process in two phases. The full boot authentication phase affirms trust in the environment and triggers a new setup phase as an anti-tampering countermeasure. While the environment is trusted, a first phase sampled boot authentication verifies the integrity of the system with a degree of confidence. The coupling of both phases allows the system to prevent undetected data tampering while meeting the constrained availability requirements of embedded systems. The approach allows time constrained embedded systems to utilize the security guarantees of secure boot without the downside of

delayed startup time. It also enables, our security monitoring system to detect flash tampering in a fraction of the time that it would take with traditional authentication methods. Furthermore, the approach was evaluated both in simulation and on a target showing its efficacy and practicality. We recognize that further optimizations are possible especially when a DMAC is available which could further improve the boot timing with our approach. This is the basis for future work in which we intend to investigate ways to optimize performance while increasing sensitivity to tampering.

CHAPTER VII

Conclusion

In this work, we started with a systematic approach to safety and security co-engineering to uncover security threats that impact safety and take action within the safety constraints of the system. We then applied this approach to AUTOSAR which is the most widely used standard automotive software platform. This lead to uncovering several security gaps in the platform. To cope with these deficiencies, we proposed an HSM based security monitoring system(SecMonQ), made up of four monitors that handle the most severe attack classes in relation to safety. SecMonQ was then shown to improve the security of AUTOSAR based systems in the face of popular attacks such as CAN masquerading. To supplement SecMonQ's firmware integrity monitor and reduce the impact of secure boot on real-time systems, we introduced a probabilistic boot authentication approach. The approach significantly reduced the time before real-time systems can become available without sacrificing the need to establish trust first in the application software. In each step of this research, the safety implications of attacks and security countermeasures were studied. When an attack was detected, we proposed practical means for bringing the system to a safe state while harmonizing the safety and security goals. The work performed in this dissertation pays special attention to the unique constraints and needs of automotive systems and avoids introducing solutions that result in prohibitive cost, major impact to legacy

code, violation of safety standards, and the need for increased computational and memory resources. This makes the methods and solutions presented here practical and effective in improving the resilience of automotive safety-critical systems against security attacks. Our hope is that the work presented in this dissertation will enable further research in the area of improved hardware and software architectures to support safety-critical systems in coping with the security threats that are only expected to grow in the future.

BIBLIOGRAPHY

[1] Autosar core and premium members list. https://www.autosar.org/about/current-partners/. Accessed: 2018-07-22.

[2] Eb tresos autocore. https://www.elektrobit.com/products/ecu/eb-tresos/autocore/. Accessed: 2018-07-22.

[3] Healing vulnerabilities to enhance software security and safety. https://bit.ly/2ndStzy. Accessed: 2018-07-22.

[4] Renesas icum firmware. https://bit.ly/2LX5492. Accessed: 2018-07-22.

[5] *Specification of Core Test*. AUTOSAR Release 4.2.2.

[6] *Specification of Crypto Service Manager*. AUTOSAR Release 4.2.2.

[7] *Specification of Diagnostic Communication Manager*. AUTOSAR Release 4.2.2.

[8] *Specification of Diagnostic Event Manager*. AUTOSAR Release 4.2.2.

[9] *Specification of Flash Test*. AUTOSAR Release 4.2.2.

[10] *Specification of Module Secure Onboard Communication*. AUTOSAR Release 4.2.2.

[11] *Specification of Operating System*. AUTOSAR Release 4.2.2.

[12] *Specification of RAM Test*. AUTOSAR Release 4.2.2.

[13] *Specification of SW-C End-to-End Communication Protection Library*. AUTOSAR Release 4.2.2.

[14] *Specification of Watchdog Manager*. AUTOSAR Release 4.2.2.

[15] ISO 14229-1, road vehicles - unified diagnostic services (UDS) – part 1: Specification and requirements, 2006.

[16] *Specification of Communication*. AUTOSAR Release 4.2.2, 2015.

[17] Road vehicles – functional safety, ISO26262, 2018.

[18] https://www.infineon.com/cms/en/product/transceivers/automotive-transceiver/automotive-can-transceivers, Accessed: 2019-05-27.

[19] https://www.renesas.com/eu/en/doc/products/mpumcu/apn/rl78/001/R01AN2535ED0202-CAN.pdf, Accessed:2019-05-07.

[20] https://ariloutech.com/solutions/in-vehicle-intrusion-detection-and-prevention-system-idps/, Accessed:2019-05-07.

[21] Rh850 evaluation platform. https://www.renesas.com/eu/en/doc/products/tool/doc/004/r20ut4009ed0100_rh850f1x.pdf, Accessed:2019-05-07.

[22] Greenhills compiler. https://www.ghs.com/products/compiler.html, Accessed:2019-06-01.

[23] Abad, F. A. T., Van Der Woude, J., Lu, Y., Bak, S., Caccamo, M., Sha, L., Mancuso, R., and Mohan, S. On-chip control flow integrity check for real time embedded systems. In *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2013 IEEE 1st International Conference on* (2013), IEEE, pp. 26–31.

[24] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security* (2005), ACM, pp. 340–353.

[25] Administration, N. H. T. S., et al. Cybersecurity best practices for modern vehicles. *Report No. DOT HS 812* (2016), 333.

[26] Amorim, T., Martin, H., Ma, Z., Schmittner, C., Schneider, D., Macher, G., Winkler, B., Krammer, M., and Kreiner, C. Systematic pattern approach for safety and security co-engineering in the automotive domain. In *International Conference on Computer Safety, Reliability, and Security* (2017), Springer, pp. 329–342.

[27] Arthur, W., and Challener, D. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, 2015.

[28] Ateniese, G., Di Pietro, R., Mancini, L. V., and Tsudik, G. Scalable and efficient provable data possession. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks* (2008), ACM, p. 9.

[29] Automotive, E. Hardware security modules unleash autosar. *EDN Network* (2018).

[30] Avatefipour, O., Hafeez, A., Tayyab, M., and Malik, H. Linking received packet to the transmitter through physical-fingerprinting of controller area network. *arXiv preprint arXiv:1801.09011* (2018).

[31] Aven, T. A unified framework for risk and vulnerability analysis covering both safety and security. *Reliability engineering & System safety 92*, 6 (2007), 745–754.

[32] Bai, Y. *Practical Microcontroller Engineering with ARM Technology*. John Wiley & Sons, 2015.

[33] Barker, E., and Kelsey, J. Nist special publication 800-90a revision 1: Recommendation for random number generation using deterministic random bit generators. *NIST, June 20q5, http://nvlpubs. nist. gov/nistpubs/SpecialPublications/NIST. SP* (2015).

[34] Böhner, M., Mattausch, A., and Much, A. Extending software architectures from safety to security. *Automotive-Safety & Security 2014* (2015).

[35] Broy, M. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering* (2006), ACM, pp. 33–42.

[36] Bubeck, O., Gramm, J., and Ihle, M. A hardware security module for engine control units. In *escar -Embedded Security in Car* (Dresden, Germany, Nov. 2011).

[37] Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR) 50*, 1 (2017), 16.

[38] Burton, S., Likkei, J., Vembar, P., and Wolf, M. Automotive functional safety= safety+ security. In *Proceedings of the First International Conference on Security of Internet of Things* (2012), ACM, pp. 150–159.

[39] Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T., et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium* (2011), San Francisco.

[40] Chen, K., Zhang, S., Li, Z., Zhang, Y., Deng, Q., Ray, S., and Jin, Y. Internet-of-things security and vulnerabilities: Taxonomy, challenges, and practice. *Journal of Hardware and Systems Security 2*, 2 (2018), 97–110.

[41] Cho, K.-T., and Shin, K. G. Fingerprinting electronic control units for vehicle intrusion detection. In *25th {USENIX} Security Symposium ({USENIX} Security 16)* (2016), pp. 911–927.

[42] Christoulakis, N., Christou, G., Athanasopoulos, E., and Ioannidis, S. Hcfi: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (2016), ACM, pp. 38–49.

[43] Committee, S. V. E. S. S., et al. Sae j3061-cybersecurity guidebook for cyber-physical automotive systems. *SAE-Society of Automotive Engineers* (2016).

[44] Corbett, C., Brunner, M., Schmidt, K., Schneider, R., and Dannebaum, U. Leveraging hardware security to secure connected vehicles. Tech. rep., SAE Technical Paper, 2018.

[45] Davi, L., Koeberl, P., and Sadeghi, A.-R. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE* (2014), IEEE, pp. 1–6.

[46] Davi, L., and Sadeghi, A.-R. *Building Secure Defenses Against Code-Reuse Attacks*. Springer, 2015.

[47] Day, D. J., and Zhao, Z.-X. Protecting against address space layout randomisation (aslr) compromises and return-to-libc attacks using network intrusion detection systems. *International Journal of Automation and Computing 8*, 4 (2011), 472–483.

[48] Dwoskin, J. S., Gomathisankaran, M., Chen, Y.-Y., and Lee, R. B. A framework for testing hardware-software security architectures. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010), ACM, pp. 387–397.

[49] Foster, J. C., Osipov, V., Bhalla, N., and Heinen, N. Buffer overflow attacks: Detect, exploit. *Prevent. Syngress Publishing* (2005).

[50] Francillon, A., and Castelluccia, C. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 15–26.

[51] Francillon, A., Perito, D., and Castelluccia, C. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code* (2009), ACM, pp. 19–26.

[52] Fuchs, A., Krauss, C., and Repp, J. Advanced remote firmware upgrades using tpm 2.0. In *IFIP International Conference on ICT Systems Security and Privacy Protection* (2016), Springer, pp. 276–289.

[53] Glas, B., Gebauer, C., Hänger, J., Heyl, A., Klarmann, J., Kriso, S., Vembar, P., and Wörz, P. Automotive safety and security integration challenges. In *Automotive-Safety & Security* (2014), pp. 13–28.

[54] Greenberg, A. Hackers cut a corvette's brakes via a common car gadget. *Wired* (2015).

[55] Hartwich, F. Controller area network with flexible data-rate, May 24 2016. US Patent 9,350,617.

[56] Hoeller, A., and Toegl, R. Trusted platform modules in cyber-physical systems: On the interference between security and dependability. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (2018), IEEE, pp. 136–144.

[57] II, O. Scan tool equivalent to iso/dis 15031-4, sae std. *J1978 199* (2001), 203.

[58] Ji, H., Wang, Y., Qin, H., Wang, Y., and Li, H. Comparative performance evaluation of intrusion detection methods for in-vehicle networks. *IEEE Access 6* (2018), 37523–37532.

[59] Johnson, C. Cybersafety: on the interactions between cybersecurity and the software engineering of safety-critical systems. *Achieving System Safety* (2012), 85–96.

[60] Kaja, N., Nasser, A., Ma, D., and Shaout, A. Automotive security. In *Encyclopedia of Wireless Networks* (2018), Springer.

[61] Karthik, T., Brown, A., Awwad, S., McCoy, D., Bielawski, R., Mott, C., Lauzon, S., Weimerskirch, A., and Cappos, J. Uptane: Securing software updates for automobiles. In *International Conference on Embedded Security in Car* (2016), pp. 1–11.

[62] Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., and Ponomarev, D. Branch regulation: Low-overhead protection from code reuse attacks. In *ACM SIGARCH Computer Architecture News* (2012), vol. 40, IEEE Computer Society, pp. 94–105.

[63] Kelion, L. Nissan leaf electric cars hack vulnerability disclosed. *BBC News. Np 24* (2016).

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

[65] Kriaa, S., Pietre-Cambacedes, L., Bouissou, M., and Halgand, Y. A survey of approaches combining safety and security for industrial control systems. *Reliability engineering & system safety 139* (2015), 156–178.

[66] Lab, T. K. S. Experimental security research of tesla autopilot. `https://keenlab.tencent.com/en/whitepapers/Experimental_Security_Research_of_Tesla_Autopilot.pdf`, Accessed:2019-04-20.

[67] Lee, J., Heo, I., Lee, Y., and Paek, Y. Efficient security monitoring with the core debug interface in an embedded processor. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 22*, 1 (2016), 8.

[68] Litzenberger, D. C. Pycrypto-the python cryptography toolkit. *URL: https://www. dlitz. net/software/pycrypto* (2016).

[69] Löhr, H., Sadeghi, A.-R., and Winandy, M. Patterns for secure boot and secure storage in computer systems. In *2010 International Conference on Availability, Reliability and Security* (2010), IEEE, pp. 569–573.

[70] Ltd., A. Trustzone technology for armv8-m architecture. `https://developer.arm.com/ip-products/security-ip/trustzone`, Accessed:2019-05-07.

[71] Macher, G., Sporer, H., Berlach, R., Armengaud, E., and Kreiner, C. Sahara: a security-aware hazard and risk analysis method. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition* (2015), EDA Consortium, pp. 621–624.

[72] Martin, B., Brown, M., Paller, A., Kirby, D., and Christey, S. 2011 cwe/sans top 25 most dangerous software errors. *Common Weakness Enumer 7515* (2011).

[73] Miller, C., and Valasek, C. Adventures in automotive networks and control units. *DEF CON 21* (2013), 260–264.

[74] Miller, C., and Valasek, C. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA* (2015), 91.

[75] Müter, M., and Asaj, N. Entropy-based anomaly detection for in-vehicle networks. In *2011 IEEE Intelligent Vehicles Symposium (IV)* (2011), IEEE, pp. 1110–1115.

[76] Nasser, A., and Gumise, W. Authenticated boot acceleration algorithm. doi: `http://dx.doi.org/10.7302/yeh1-1x17`.

[77] Nasser, A., Gumise, W., and Ma, D. Accelerated secure boot for real-time embedded safety systems. *SAE International Journal of Transportation Cybersecurity and Privacy 2* (2019). doi:`10.4271/11-02-01-0003`.

[78] Nasser, A., and Ma, D. Defending AUTOSAR safety critical systems against code reuse attacks. In *Proceedings of the ACM Workshop on Automotive Cybersecurity* (2019), ACM, pp. 15–18.

[79] Nasser, A., Ma, D., and Lauzon, S. Safety-driven cyber security engineering approach applied to OTA. In *Computer Science,Computer Engineering and Applied Computing* (July 2016).

[80] Nasser, A. M., and Ma, D. SecMonQ: An HSM Based Security Monitoring Approach for Protecting AUTOSAR Safety-critical Systems. *Vehicular Communications* (2019), 100201. doi:`10.1016/j.vehcom.2019.100201`.

[81] Nasser, A. M., Ma, D., and Lauzon, S. Exploiting AUTOSAR safety mechanisms to launch security attacks. In *International Conference on Network and System Security* (2017), Springer, pp. 73–86.

[82] Nilsson, D. K., Larson, U. E., and Jonsson, E. Creating a secure infrastructure for wireless diagnostics and software updates in vehicles. In *International Conference on Computer Safety, Reliability, and Security* (2008), Springer, pp. 207–220.

[83] Paulitsch, M., Morris, J., Hall, B., Driscoll, K., Latronico, E., and Koopman, P. Coverage and the use of cyclic redundancy codes in ultra-dependable systems. In *2005 International Conference on Dependable Systems and Networks (DSN'05)* (2005), IEEE, pp. 346–355.

[84] Pearson, S. Trusted computing platforms, the next security solution. *HP Labs* (2002).

[85] Pièvre-Cambacédès, L. *Des relations entre sûreté et sécurité*. PhD thesis, Télécom ParisTech, 2010.

[86] Ravi, S., Raghunathan, A., Kocher, P., and Hattangady, S. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems (TECS) 3*, 3 (2004), 461–491.

[87] Ruddle, A., Ward, D., Weyl, B., Idrees, S., Roudier, Y., Friedewald, M., Leimbach, T., Fuchs, A., Gürgens, S., Henniger, O., et al. Deliverable d2. 3: Security requirements for automotive on-board networks based on dark-side scenarios. *tech. rep., EVITA* (2009).

[88] Sha, K., Wei, W., Yang, T. A., Wang, Z., and Shi, W. On security challenges and open issues in internet of things. *Future Generation Computer Systems 83* (2018), 326–337.

[89] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 552–561.

[90] Shepherd, C., Arfaoui, G., Gurulian, I., Lee, R. P., Markantonakis, K., Akram, R. N., Sauveron, D., and Conchon, E. Secure and trusted execution: Past, present, and future-a critical review in the context of the internet of things and cyber-physical systems. In *2016 IEEE Trustcom/BigDataSE/ISPA* (2016), IEEE, pp. 168–177.

[91] Shostack, A. Experiences threat modeling at microsoft. In *MODSEC@ MoDELS* (2008).

[92] Smith, C. *The car hacker's handbook: a guide for the penetration tester*. No Starch Press, 2016.

[93] Song, H. M., Kim, H. R., and Kim, H. K. Intrusion detection system based on the analysis of time intervals of can messages for in-vehicle network. In *2016 international conference on information networking (ICOIN)* (2016), IEEE, pp. 63–68.

[94] Song, J., Poovendran, R., Lee, J., and Iwata, T. The aes-cmac algorithm. Tech. rep., 2006.

[95] Stoneburner, G. Toward a unified security-safety model. *Computer 39*, 8 (2006), 96–97.

[96] Taylor, A., Japkowicz, N., and Leblanc, S. Frequency-based anomaly detection for the automotive can bus. In *2015 World Congress on Industrial Control Systems Security (WCICSS)* (2015), IEEE, pp. 45–49.

[97] TCG TPM 2.0 Automotive Thin Profile For TPM Family 2.0. Specification, Trusted Computing Group, May 2018.

[98] Tencent. New car hacking research: 2017, remote attack tesla motors again. *Keen Security Lab Blog* (2017).

[99] van der Veen, V., Andriesse, D., Göktaş, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., and Giuffrida, C. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 927–940.

[100] Weyl, B., Wolf, M., Zweers, F., Gendrullis, T., Idrees, M. S., Roudier, Y., Schweppe, H., Platzdasch, H., El Khayari, R., Henniger, O., et al. Secure on-board architecture specification. *Evita Deliverable D 3* (2010), 2.

[101] Wiersma, N., and Pareja, R. *A security assessment of the resilience against fault injection attacks in ASIL-D certified microcontrollers.* esCar, 2017.

[102] Wolf, T., Mao, S., Kumar, D., Datta, B., Burleson, W., and Gogniat, G. Collaborative monitors for embedded system security. In *1st Workshop on Embedded Systems Security (EMSOFT). ACM* (2006), Citeseer.

[103] Zhang, T., Zhuang, X., Pande, S., and Lee, W. Anomalous path detection with hardware support. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems* (2005), ACM, pp. 43–54.