An Approach for Building Security Resilience in AUTOSAR Based Safety Critical Systems

Ahmad MK Nasser[1], Di Ma[1] and Priya Muralidharan[2]

[1]*University of Michigan, Dearborn, USA*
[2]*Renesas Electronics America*
E-mail: {ahmadnas,dmadma}@umich.edu; priya.muralidharan@renesas.com

Received 5 December 2017; Accepted 6 December 2017;
Publication 19 December 2017

Abstract

AUTOSAR, a worldwide development partnership among automotive parties to establish an open and standardized software architecture for electronic control units (ECUs), has seen great success in recent years by being widely adopted in deeply embedded automotive ECUs. Increasing the security resilience of AUTOSAR based systems is a crucial step in securing safety critical automotive systems. We study AUTOSAR safety mechanisms and demonstrate how they can be used as attack vectors to degrade the vehicle safety. We show the need to harmonize the fail-safe response with the secure state of the system. And we evaluate the overlap in the properties of safety mechanisms with security objectives to highlight methods for hardening automotive systems security.

Keywords: Automotive cyber security, AUTOSAR, Safety, Security, ISO26262, Embedded security.

1 Introduction

Protecting the vehicle control side against cyber attacks is the ultimate goal of any robust security architecture. With added connectivity, even deeply embedded devices are under the threat of cyber attacks [23, 24, 28].

The safety standard for automotive systems, ISO26262 [26], defines a formal approach for the creation of safety mechanisms to mitigate systematic and random faults in the system. However, malicious attacks are not considered by the failure response of the system. When a safety relevant failure condition is detected, the system switches to a safe state, also known as a fail-safe state. Depending on the risk level associated with the related hazard, a safe state may range from disabling a control function (e.g. park assist in the case of a sensor failure), to resetting the system (e.g. in the case of detected memory corruption). Safety mechanisms which are adequate in the face of normal failures, are inadequate and in some cases, even themselves exploitable, in the presence of a malicious attacker [25]. Injecting a safety fault can thus launch the equivalent of DoS attack against the vehicle safety functions. While there is a real need for hardening the security of deeply embedded systems, it has to be done within the cost and performance constraints of automotive systems.

The need for security-in-depth measures for securing automotive systems is well-understood. At the ECU level, this means participating in secure communication, supporting for code signing of flash downloads, securing the diagnostic interface, securing the debug interfaces, and using secure boot. Even as the automotive industry takes concrete steps to apply such best practices, software bugs are still expected to present major challenges to comprehensive vehicle security. For security, AUTOSAR defines a cryptographic security stack to provide fundamental security services. It also defines safety mechanisms with overlapping security properties. The goal of this research is to study AUTOSAR safety mechanisms as potential sources of attacks. Also to answer the question of how shall a system differentiate a safety failure which happens due to a deterministic fault vs. a failure injected through an attack in order to harmonize the safe and secure response. Moreover, is it possible to leverage existing AUTOSAR safety mechanisms with security properties to increase the system security? By studying this area, we aim to improve the security resilience of AUTOSAR based systems within the constraints of the automotive environment.

Toward this goal, we make the following contributions: we take a systematic approach to evaluate safety mechanisms that could be potentially exploited, we demonstrate two such attacks on a safety qualified target, and then propose corresponding countermeasures that can be applied for similar types of attacks. We also experiment with using the AUTOSAR OS stack interference protection as a security mechanism to prevent a buffer overflow attack. We show how AUTOSAR safety mechanisms can be used as security mechanisms and we define the constraints to enable such use. We also demonstrate scenarios in which the security response in the presence of an attack can violate the security response and vice versa to highlight the need for a harmonized process for fail safe and secure action.

The organization of this paper is as follows. In Section 2, we briefly discuss related work. In Section 3, we present the attack model considered for our analysis. In Section 4, we offer a survey of AUTOSAR safety features. In Section 5, we present a method for assessing safety mechanisms as either exploitable or dually useful for security protection. In Section 6, we analyze safety mechanisms as exploitable candidates. In Section 7, we analyze safety mechanisms that have overlapping security properties. In Section 8, we present the results of our attacks on safety mechanisms, while in Section 9 we demonstrate the efficacy of a safety mechanism as a security mechanism. In Section 10 is the conclusion and future work.

2 Related Work

The use of an automotive error mitigation mechanism as a security attack vector was studied in [13]. The authors demonstrated an attack that uses the CAN busoff error response to stop an ECU from communicating. The Bus-Off condition is designed to prevent an ECU from disturbing the rest of the bus by forcing it to go off line after a certain number of transmission errors is detected. By creating message collisions with a target CAN frame an attacker could induce a Bus-Off error condition in the target ECU. This is similar to an exploitable safety mechanism but we take a systematic approach to finding such exploits and we do it by studying AUTOSAR safety mechanisms.

The conflicts and overlap between safety and security for automotive systems was studied in [21]. The work highlighted the need for a holistic approach to designing both safe and secure systems. One example conflict area is the cyclic RAM test that requires stopping CPU cores except the one performing the test in order to prevent a false detection of RAM corruption. In systems where a hardware security module (HSM) is active, stopping the HSM violates a security principle. Not stopping the HSM can result in corruption of the shared memory accessible by the HSM and the other CPUs. This points to an area of conflict that should be addressed at the system level. The paper also discusses areas of overlap between safety and security such as using the memory protection unit (MPU) to provide both freedom from interference as well as security isolation. Our goal in this paper is to increase the resilience of AUTOSAR systems by eliminating exploits and adding defenses strictly from the perspective of safety.

Attacks against embedded systems were studied in [18]. The authors demonstrated control flow and stack overflow attacks on embedded systems and then proposed a hardware based solution for the prevention of such attacks. Their attacks were relevant to our work as we aim to provide defense mechanisms for embedded systems against the types of attacks presented in their work.The use of the CPU dual lock step safety mechanism as a security countermeasures was studied in [30]. The authors investigated the resilience of ASIL-D microcontrollers to power glitching attacks. They showed that the dual lock step mechanism failed to detect glitch attacks allowing them to jump over instructions and thus compromising the security of an ECU. The paper shined a light on how safety mechanisms needed to evolve to protect against security attacks. Our research takes this concept a step further by performing a comprehensive survey of AUTOSAR safety mechanisms to find areas in which safety mechanisms could be useful to increase the security of safety critical systems.

3 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 [24] and [28], have shown that at some point these defenses can be broken and an attacker may be able to launch a successful attack on the vehicle control system. Let us consider the three classes of attacks that are relevant to both deeply embedded ECUs and traditional computer systems [15]:

1. Malware or exploitable software vulnerabilities to take control of the system
2. Physical access type attacks
3. Network based attacks

Note in the first class, there is a significant difference in the attack surface between traditional computers and deeply embedded systems [12]. 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.

On one hand, 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 [17]. When network access is considered together with software based exploits, deeply embedded systems become targets of 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 cyber security principles within Automotive ECUs gives us the intuition that software vulnerabilities and back doors will persist for several years. On the other hand, in case of ADAS systems which are expected to provide a certain level of autonomy, such systems may no longer exist in a deeply embedded layer, but rather in a mixed criticality system on rich execution environments which are closer to the attackers access than say a brake controller. For such systems, the safety functions are expected to run in a partitioned execution area that is supposedly isolated from the rest of the system. That is the focus of AUTOSAR Adaptive Platform [19] and is out of scope for our current analysis.

For the rest of this paper we assume our attackers have direct network access to the ECU, the network supports CAN authentication, and there exists an exploit in the ECU which allows loading software on the target. We also assume that the ECU has a secure element (HSM) which serves as the root of trust for the system. As for the attacker's objective, it is to disrupt safety critical systems for the aim of inflicting harm on drivers, damaging an OEM reputation, or creating ransom-ware that locks up safety functions.

4 Survey of AUTOSAR Safety Features

4.1 End to End Library

The first AUTOSAR module that we introduce here is the End to End (E2E) library which defines several protection profiles for data transmitted over a communication channel both internally and externally [20]. The goal of this module is to prevent safety critical functions from operating on faulty or missing data as shown below:

1. CRC check to detect corruption of data
2. Sequence counter to detect out of sequence messages
3. Alive counter to prevent operating on old data
4. A unique ID for Interaction Layer Protocol Data Unit (I-PDU) group to detect a fault of sending I-PDU on unintended message
5. Timeout monitoring to detect communication loss with the sender

Mechanisms 1 through 4 listed above allow the detection of non-malicious errors in the content of a message. The timeout detection mechanism protects an ECU from operating on old data due to loss of communication with the data source. To achieve this, a message is monitored by the receiving node based on its expected periodicity. If the message is not received by the expected deadline, AUTOSAR provides a mechanism to log a timeout failure and take fail safe action such as disabling the consuming function.

4.2 AUTOSAR OS

AUTOSAR OS [7] is a real time operating system specification. The standard defines protection mechanisms that are essential for building safety critical applications. Those protections fall under the following categories:

1. Memory Protection: to provide freedom of interference between OS applications and tasks of mixed criticality.
2. Timing Protection: to prevent timing errors in tasks, Interrupt Service Routines (ISRs), or system resource locks from interfering with higher ASIL functions.
3. Service Protection: to capture invalid use of the OS services by the application.
4. 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:

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

4.2.1 Memory protections

AUTOSAR OS SC3 and SC4 support freedom of interference between software partitions of mixed safety criticality through hardware based spatial separation of memory [7]. The aim of these protections is to prevent a 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 [11]. Rather than an MMU, such MCUs rely on an MPU to achieve the desired goal of freedom from interference. 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 Cat2 level and at the OS application level. When switching to a "non-trusted" OS application, the OS reconfigures the MPU to restrict access to the Safety Application code, data and private stack. This prevents a lower ASIL OS application from being able to corrupt 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 only identify where a task or ISR has exceeded a specified stack usage at context switch time. This is done by checking a unique stack pattern which is inserted at the end of the reserved stack space. This can result in considerable time between the system being in error and the fault being detected. 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. 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.

4.2.2 Timing protections

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

1. Execution Time Protection: detects faults in Tasks or Category 2 ISRs that exceed their execution budget.
2. Locking Time Protection: detects faults in blocking resources, and locking interrupts for a period longer than the configured maximum.
3. Inter-arrival time protection: detects faults in the time between successive activations of tasks or Cat2 interrupts to ensure a minimum time is not violated.

By monitoring the execution time of tasks and Category 2 ISRs, AUTOSAR OS aims to 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. Note the OsTaskExecutionBudget is a configurable parameter that specifies the maximum allowed execution time of a task [7]. 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 the real time system. Disabling global interrupts is needed in scenarios where a routine needs atomic access to a resource and cannot allow 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 or activating tasks.

4.3 Hardware Protection

AUTOSAR OS is expected to run in privileged mode which gives it access to special hardware registers and protects those registers 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.

4.4 Watchdog Manager

AUTOSAR defines three modules for supporting watchdog functions [9]:

1. Watchdog Driver: services the hardware watchdog whether internal or external
2. Watchdog Interface: provides a high level of abstraction of watchdog driver functions
3. 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 and they are marked by checkpoints. The user configures checkpoints based on code sections which are deemed to be critical for the safety of the system. Using such checkpoints it is possible to monitor aliveness, timeout, and control flow within a supervised entity.

The user inserts calls to WdgM_CheckpointReached() in locations where the WdgM shall be notified of an execution event of a supervised entity.

A software error that prevents the checkpoint from being reached by the deadline or with the right order 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 notification to an MCU reset.

4.5 Core Test

Safety critical applications require the monitoring of an MCU core functions to detect hardware faults during startup or normal runtime of an ECU. The core test module [2], can perform tests of MCU components such as:

1. Arithmetic Logic Unit (ALU)
2. Memory Protection Unit (MPU)
3. Cache controller
4. 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) [4] to take action based on the severity of the detected failure. Note that microcontrollers 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.

4.6 RAM Test

The RAM test module [8] 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 set the test status to a 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:

1. Checkerboard test algorithm
2. March test algorithm
3. Walk path test algorithm
4. Galpat test algorithm
5. Transparent Galpat test algorithm
6. Abraham test algorithm

4.7 Flash Test

The Flash Test module [5], 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 RAM test and Core Test, 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:

1. 16 Bit CRC
2. 32 Bit CRC
3. 8 Bit CRC
4. Checksum
5. Duplicated Memory
6. ECC

5 Safety Mechanism Classification Method

We propose a method for classifying AUTOSAR safety mechanisms as either candidates for: security exploits, security protections, or neither. In Figure 1, we present the process of finding an exploit candidate. Note, here the attacker's intent is to inject a failure that mimics a safety failure in order to shutdown the safety function. First we check if the mechanism can be triggered through a network based attack. Intuitively, an attacker that has already established a foothold in the ECU has very little incentive to trigger the error response of a safety mechanism. Arguably, once an attacker is inside the ECU, his goal is to evade any protection mechanisms which is the opposite of triggering a safety mechanism. Next we check if the configured error response of the safety mechanism results in disabling a safety function or more (such as a system reset). If so, we classify this mechanism as an exploitation candidate. Here

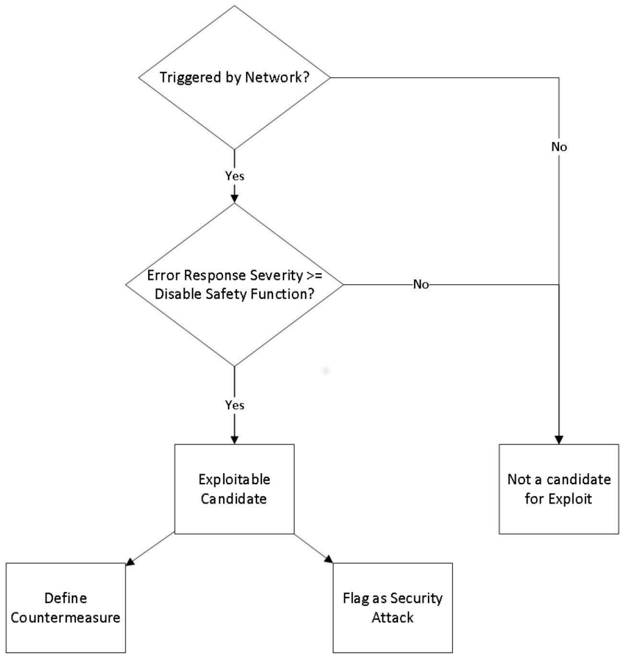


Figure 1 Method to classify a safety mechanism as a candidate for a security exploit.

we define two possible routes: define a security countermeasure to prevent the attack, and for cases where prevention is not possible, extend the fail-safe response to capture additional indicators that can help in flagging this event as a security attack.

In Figure 2, we present the process for classifying a safety mechanism as a dual purpose security mechanism. First we check if the mechanism is able to detect an attack. This can be an internal attack through malware, a physical attack like power glitching, or a network attack. Safety mechanisms that exhibit properties to allow memory isolation, hardware resource protection, control flow protection and hardware tampering detection are all example candidates for use as security mechanisms. But having a security property is not sufficient if the mechanism can be easily disabled by an attacker. Therefore we define a set of security constraints to establish trust in the usage of the safety mechanism as an attack prevention or detection mechanism. The third step is to ensure that the fail-safe response is harmonized with the required security response and vice versa. For example, in case of a mechanism that detects the presence of a malware in RAM, the security response would be to immediately

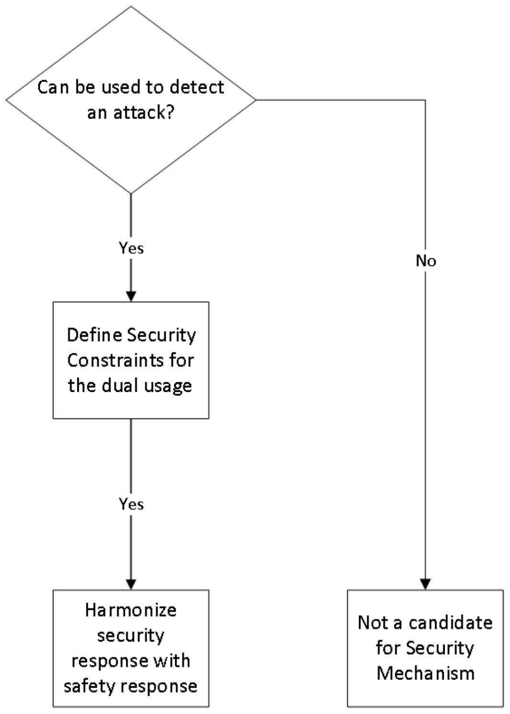

Can be used to detect an attack?
Yes
No
Define Security Constraints for the dual usage
Yes
Harmonize security response with safety response
Not a candidate for Security Mechanism

Figure 2 Method to classify a safety mechanism as a dual purpose security mechanism.

issue a reset and force a secure boot check to restore the system to a known trusted state. But issuing an immediate reset may violate a safety goal that does not allow the sudden loss of a critical safety function, for e.g. power steering. Therefore, an activity of harmonizing the safe and secure response is needed between the safety and security experts during the design of the system.

In the following sections, we apply these two methods to classify our surveyed safety features.

6 Exploitable Safety Mechanisms

This section was based on our work in [25]. There we evaluated safety mechanisms in the AUTOSAR E2E library, and AUTOSAR OS timing protections as security exploit candidates. With the E2E library, our aim was to induce a failure that would trigger a protection mechanism which would result in disabling safety critical functionality in an ECU. With AUTOSAR

OS timing protections, our aim was to trigger a system shutdown by causing a software task to exceed its runtime budget.

6.1 Attacks on E2E Protection

We start by applying the classification flow in Figure 1 to the safety mechanisms in Table 1. We assume that the network supports message authentication which can protect against spoofing attacks. This prevents an attacker from tampering with the alive counter, CRC, sequence counter or I-PDU Id protection mechanism without detection because they are covered by the message authentication code(MAC). Messages that contain invalid MAC are ignored by the receiver and therefore do not result in a trigger of a safety mechanism. What remains then is the timeout protection mechanism which can be triggered externally even when message authentication is enabled. 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.

Table 1 Survey of AUTOSAR safety features

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	MCU reset
WdgM	Detect timeout of supervised entity	MCU resets
WdgM	Monitor control flow of supervised entity	MCU resets
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

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.

6.1.1 Countermeasures

In [25] we showed several countermeasures:

1. Add a smart gateway that runs intrusion detection software to monitor the received CAN message identifiers along with their expected frequency to detect attacks such as the zero-ID flood attack.
2. Add local secure monitor software within transmitting ECU's to detect the malicious manipulation of the CAN configuration. The monitor can either reside in the HSM or be monitored periodically by the HSM to ensure it is not disabled. If the CAN settings are flagged as tampered with, the HSM can reset the micro to 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 [24], the attack is still worth mentioning here because we arrive at it by considering the safety protection mechanisms, rather than by pure brainstorming techniques. We also stress the need to enhance the logging of such an event to identify potential malicious root causes.

6.2 Task Execution Time Attack

By applying the classification flow in Figure 1 to the safety mechanisms in Table 1 we arrive at our second exploit candidate: Task Execution Budget

Monitoring. In response to this type of timing error, the OS defines the following possible actions that the application can request [7]:

1. PRO_IGNORE: the OS can ignore the event
2. PRO_TERMINATE_TASKISR: the OS shall forcibly terminate the task
3. PRO_TERMINATE_APPL: the OS shall terminate the faulty OS Application
4. PRO_TERMINATE_APPL_RESTART: the OS shall terminate and then restart the faulty OS Application
5. PRO_SHUTDOWN: the OS shall shutdown itself

The last action from the above list implies that the system can be completely shutdown as a result of such an error condition. 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. 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 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 normally.

Upon detecting the error condition the OS triggers a ProtectionHook to notify the application to take fail safe measures. As long as the application does not ignore this condition, the second condition of Figure 1 is satisfied. For the rest of this section we will see how this fault can be triggered externally.

As mentioned in Section 4.2.2, AUTOSAR OS monitors the task execution time, to prevent a single task from starving the CPU from runtime resources. To exploit this safety mechanism, the attackers goal is to cause an OS task to exceed its execution budget. One way to find candidates for this type of exploit 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 like processing CAN data.

We chose the latter and we investigated CAN networks that support authenticated messages as proposed in AUTOSAR 4.2 via the Secure On Board Communication (SecOC) module [6]. SecOC is a software module that provides secure on board communication support. 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 3. Only if the verification passes, SecOC then notifies the

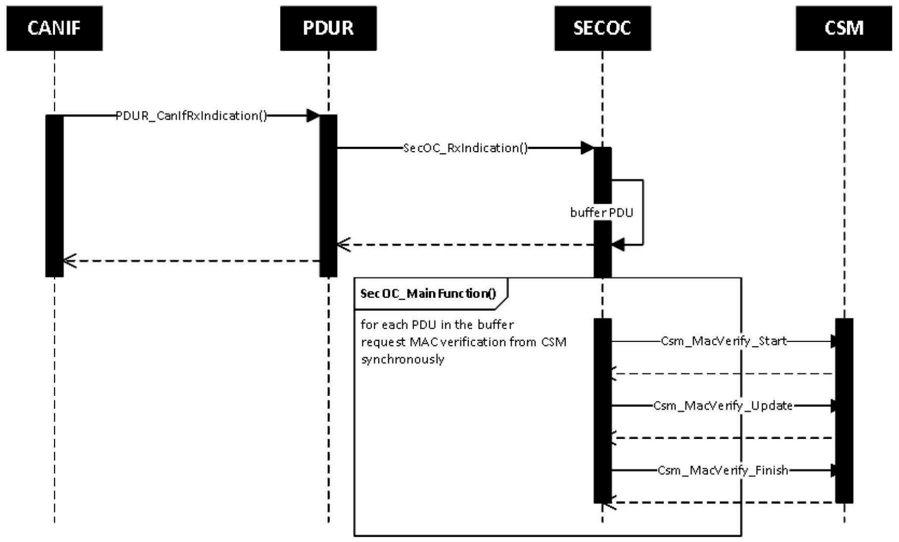


Figure 3 Sequence diagram showing data flow from CANIF to the CSM layer for frame authentication.

PDUR module to route the PDU up to the consuming layers [6]. 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 runtime budget: OsTaskExecutionBudget.

Based on the CAN FD specification [27], 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 8, 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):

$$Tcanfd = (1 + f) * (30/a + (28 + dl * 8)/d). \tag{1}$$

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 is equivalent to a factor of 20%.

In the case that CAN message authentication is enabled the attacker takes advantage of the fact that an ECU needs to spend a fixed amount of CPU runtime 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 micro-controller 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 3, SecOC_MainFunction() loops through all the buffered PDUs that require verification and triggers the verification request to the AUTOSAR CSM [3] module. We intentionally choose to configure CSM to run in synchronous mode so as to maximize 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 runtime execution budget. 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 there exists a value $B \leq maxB$ such that $T_{processing} > T_{budget}$ where:

$$maxB = \frac{T_{secoc}}{T_{canfd}}. \tag{2}$$

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 $B \leq maxB$.

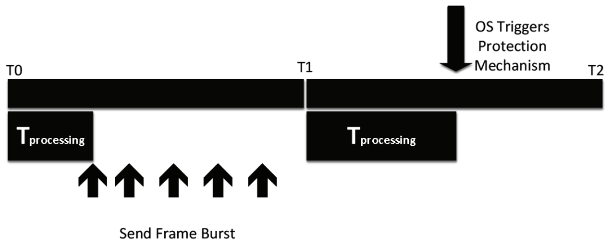


Figure 4 Triggering the OS protection mechanism by causing the SecOC main function task to exceed the execution budget.

In order to evaluate if a system is affected by this attack, we present Equation (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 runtime 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:

$$B = \frac{T_{budget}}{(N_{attempts} * (T_{main} + T_{mac}))} \tag{3}$$

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 8.2.

6.2.1 Countermeasures

The countermeasures to this attack as shown in [25] consist of the following:

1. 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.
2. Wherever possible, choose the asynchronous mode for performing specific functions. 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.
3. 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.
4. Use a network anomaly detection system that can flag and stop anomalous message bursts like the one used in this paper to trigger the attack.
5. 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 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.

7 Dual Purpose Safety Mechanisms

Being able to use AUTOSAR safety mechanisms for attack detection can increase the security resilience of AUTOSAR based systems without a major impact to the cost of the system. In the below sections we show the results of our analysis of AUTOSAR safety mechanisms as means for attack detection.

7.1 Analysis

7.1.1 Stack usage monitoring

Code injection attacks through buffer overflow [16] continue to be among the most effective in computer systems, where an attacker can overflow a buffer boundary in the task stack in order to overwrite the return address of a function. This results in rerouting the flow of the program to the new address overwritten by the overflow.

Naturally, a protection mechanism that can detect stack overflow would be relevant for security. As mentioned in Section 4.2.1, 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 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. This matches well as a mechanism for attack detection.

7.1.2 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 mentioned in Section 4.2.1 using an MPU it is possible to setup 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.

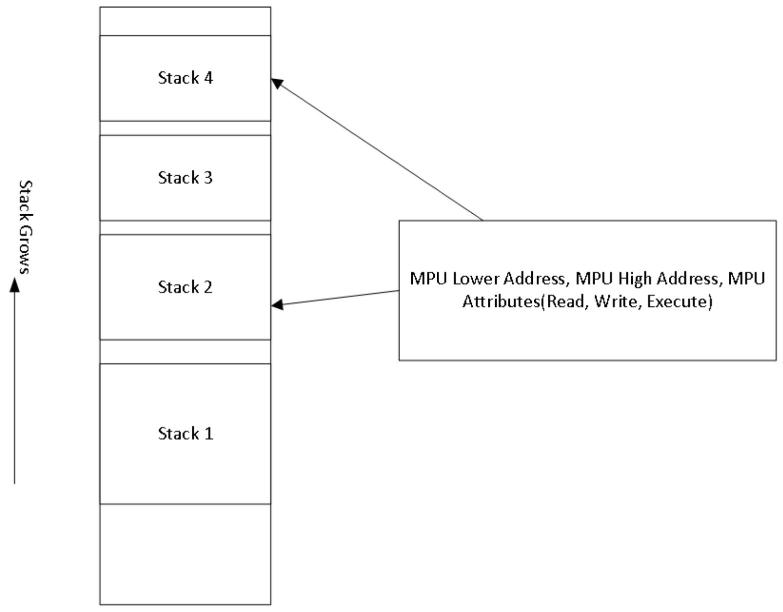

Stack 4
Stack 3
Stack 2
Stack 1
Stack Grows
MPU Lower Address, MPU High Address, MPU Attributes(Read, Write, Execute)

Figure 5 MPU based stack protection, by switching MPU setting based on the active stack.

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 [14], 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.

7.1.3 Control flow integrity protection

The topic of control flow integrity(CFI) was thoroughly studied in [10]. Moreover, [17] showed several techniques to alter the execution flow specifically in an embedded system. Having a mechanism that can enforce program flow can be useful in mitigating attacks such as return oriented programming and stack based code injection. AUTOSAR offers a mechanism that can be re-purposed for CFI, namely 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 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. Note that one can argue that an attacker can reload the watchdog timer to prevent a reset. This can be made more difficult by only using a windowed watchdog timer. This means the attacker would have to reload the timer in the right time window which is synchronized with the WdgM_MainFunction() call cycle.

7.1.4 OsTiming protections

The 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 from runtime or to exhaust stack memory resources. An example would be malicious network traffic that results in over triggering the CAN interrupt.

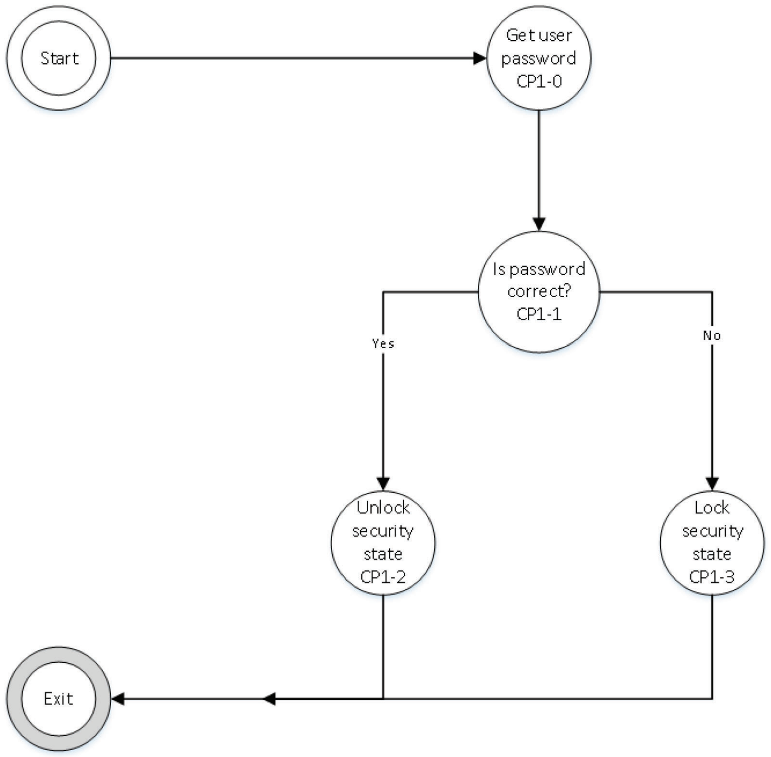

Start
Get user password CP1-0
Is password correct? CP1-1
Yes
No
Unlock security state CP1-2
Lock security state CP1-3
Exit

Figure 6 WdgM monitors if password verification checkpoint is skipped to detect an attack.

7.1.5 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 in an infected ECU.

7.2 Security Constraints

As shown in Figure 2, in order to use the safety mechanisms listed above for security, certain constraints are needed to ensure the availability and integrity of those protections.

1. Internal variables of the monitoring software shall be protected against tampering via the MPU partitioning to prevent an attacker from spoofing

their values, for e.g. WdgM global state variable which contains the status of the program flow checks.

2. Software modules executing the security monitoring and enforcement like AUTOSAR OS, WdgM and MCU Driver shall be protected against tampering via secure boot. This is to ensure that those protections are present when the system boots up.
3. Systems that allow runtime integrity checking of software can be leveraged to periodically verify the integrity and authenticity of the software modules executing the protection mechanisms.
4. Configuring OS applications as “Trusted OS Applications” shall be prohibited. The latter has full access to all memory resources which makes it a valuable target for an attacker. Moreover, “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, while all other tasks and applications shall run in user mode.
5. 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.
6. 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.
7. Any API call in AUTOSAR that allows disabling a security monitoring shall be disabled, to prevent an attacker from bypassing security checks.
8. The timer interrupt upon which the OS is relying shall be a high priority non-maskable interrupt. This ensures that the OS can execute the security monitoring functions defined in this paper.
9. When using the WdgM for control flow protection, the underlying watchdog timer shall be only configurable once after reset. Attempts to disable the watchdog shall either be ignored or result in a system reset.

7.3 Harmonizing Safety and Security Response

As shown in [29], safe degradation of a safety function can depend on the criticality of the function. Safety critical systems aim to keep alive critical functions for as long as possible when an error is detected. This leads to the definition of different degradation levels that aim to keep alive those functions that are deemed most critical to a vehicle. A function is allowed per the FTTI(fault tolerant time interval) before it has to be degraded due to a fault [27]. Furthermore, AUTOSAR provides flexibility in defining the error response which can be as harmless as a simple notification to the application. From a security point of view, detecting an attack, especially one that has compromised internal memory of the target shall result in a quick response to restore the target to a trusted state. One possible response is to clear the contents of volatile memory and issue a system reset. This can then trigger the secure boot mechanism which checks if the contents of non-volatile memory have been modified. Any such manipulation can result in locking security assets to prevent an attacker from using the affected target to launch attacks on other devices in its direct network. But a sudden reset is not acceptable for safety applications which have explicit requirements against the sudden loss of a safety critical functions like electric power steering. Therefore, there is a need to harmonize the error response to satisfy the safety and security needs of the system. This may result in creating more redundancies in the system to be able to keep alive the safety function even under an attack.

8 Attacks Evaluation

In order to evaluate the attacks described in Section 6, we build a test environment using an automotive grade 32-bit micro-controller, running 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. The identity of the micro-controller is not disclosed in this paper.

8.1 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 an RC car by translating the CAN messages into PWM signals that control the steering and driving as shown in Figure 7. 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.

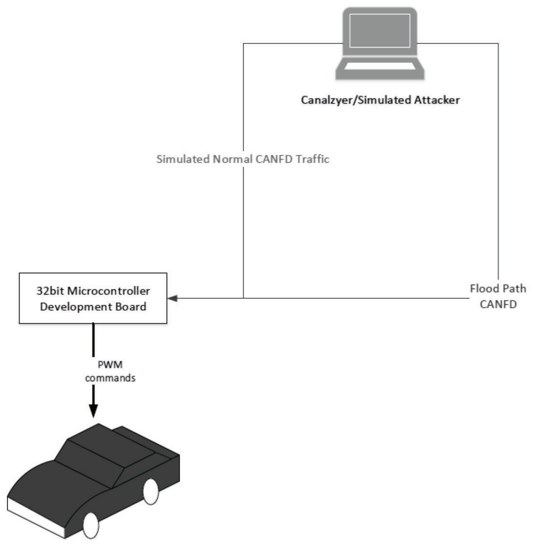

Canalzyer/Simulated Attacker
Simulated Normal CANFD Traffic
32bit Microcontroller Development Board
Flood Path CANFD
PWM commands

Figure 7 Data Architecture for zero-ID attack.

SOF	11 bit CAN Identifier	r1	IDE	EDL	r0	BRS	ESI	4bit DLC	0-64 bytes: Data Field	21 bit CRC	1	1	1	7 bit EOF	3 INT	IDLE

Figure 8 CAN FD Frame layout, for 64 byte frames CRC is 21 bits long [22].

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 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 RC car becomes very difficult as only a small fraction of control messages is transmitted on the bus. In a real vehicle with timeout monitoring protection, the receiver would simply disable the steering and driving control functions in response to losing messages which would correspond to a successful DoS attack.

8.2 Resource Exhaustion Attack

We first introduce CAN FD which is a relatively new communication protocol that extends the CAN 2.0 standard with a larger payload (up to 64 bytes) and a higher data rate (up to 8 Mbps) [27]. The protocol defines an arbitration baud rate, and a flexible data rate that can be higher than the arbitration rate. This allows CAN 2.0 frames to coexist with CAN FD frames.

In order to evaluate this attack we implemented a reduced AUTOSAR stack that performs the entire chain of CAN message reception and authentication and applied it to a CAN FD network. We assumed that the AUTOSAR CSM [3], 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 3. The process is repeated until all the buffered secure PDUs have been verified. The attacker was simulated by a software task that runs every 10 ms 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 10 ms cycle, i.e. T_{secoc} = 10 ms, and configured the CAN controller to receive a maximum of 40 unique messages on CAN channel 1. We also set the SecOCFreshnessCounterSync Attempts value to 1, because the freshness counter is sent in its entirety within the CAN FD frame. This is also meant to increase the attack difficulty, because a larger SecOCFreshnessCounterSync Attempts increases the $T_{processing}$ time needed to verify all the failed MAC values received making the attack easier to succeed. 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 48 bytes of payload data, 8 bytes freshness counter and 8 bytes 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 (1) and (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

Table 2 CAN FD frame time in µs based on 64 byte DLC

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

a 64 byte CAN FD frame length we aim to increase the attack difficulty by minimizing the maximum burst size possible within our time constraint of 10 ms. 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 10 ms Task. Thus we choose the T_{budget} to be 10% of the task cycle time which corresponds to 1 ms. 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 9, show that for $T_{processing}$ to exceed our chosen execution budget of 1 ms, it is

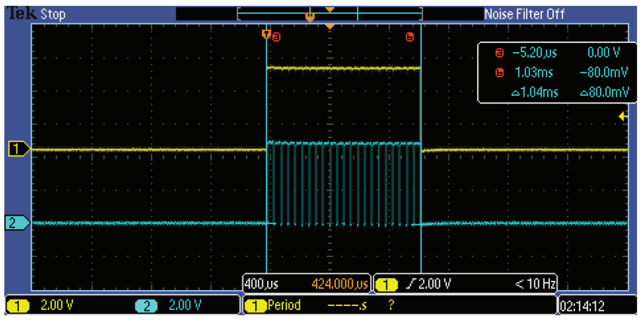


Figure 9 Total runtime is 1.04 ms for processing 22 CAN FD messages of 64 bytes each.

```
void ApplDiagWriteDataByIdentifier(uint8_t canLen)
{
        uint8_t diagBuffer[8];

        /* this routine copies data from CAN to the
        diagnostic buffer the length in the CAN
        diagnostic request is not being checked this
        allows the buffer to be overrun and the return
        address to be overwritten */

        memcpy(diagBuffer,canBuffer,canLen);
}
```

Figure 10 Buffer overflow routine.

sufficient to send a burst of 22 secure PDUs within a 10 ms cycle. Therefore the number of frames needed to trigger the AUTOSAR OS failure is below the *maxB* = 27 calculated in Table 2 which satisfies Theorem 1. Furthermore, 22 CAN messages is well within the normal number of CAN frames that a typical ECU consumes.

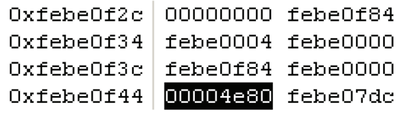
9 Defense Evaluation

For our evaluation of protection mechanisms, we chose the stack usage monitoring protection and performed the evaluation on a RH850 P1x based microcontroller [1] which is ASIL-D qualified.

9.1 Stack Overflow Protection

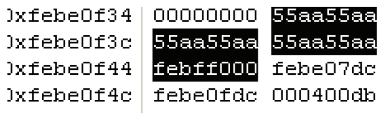
To demonstrate the security mechanism we create a CAN diagnostic vulnerability that allows a diagnostic tool to inject code into the stack memory reserved for a "trusted" OS application. The diagnostic routine that is loading the data over CAN contains a bug that allows the diagnostic tool to send a well crafted diagnostic request to overflow the stack. The received payload overwrites the return address of the calling routine with a RAM based address which corresponds to the entry point of the malicious routine. Since we do not have access to a full AUTOSAR software stack, 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 addresses to prevent an attacker from fetching instructions from the stack in the case of a successful code injection attack.

Figure 11 shows the stack state before the attack with the target address to overwrite being 0x4e80. The code listing shown in 10 is a simple routine that contains the buffer overflow exploit we used. Figure 12 shows the contents of the stack after executing the routine which causes the overwrite of the return address with 0xfebff000. Once the routine attempts to return, the CPU pulls the link pointer register value (lp) from the stack causing it to jump to the malicious code shown in Figure 14. The latter attempts to jump to the bootloader after setting the programming flag. The attacker's goal is to bypass



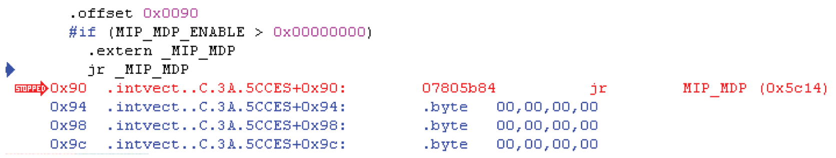
```
0xfebe0f2c  00000000 febe0f84
0xfebe0f34  febe0004 febe0000
0xfebe0f3c  febe0f84 febe0000
0xfebe0f44  00004e80 febe07dc
```

Figure 11 Stack values before executing the attack, the return address is 0x4e80.



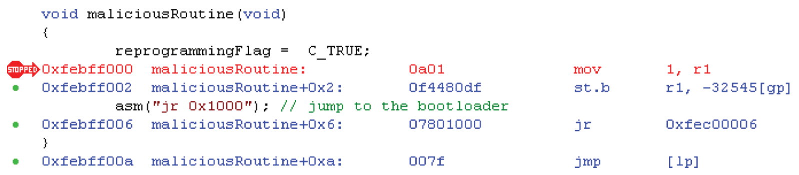
```
)xfebe0f34  00000000 55aa55aa
)xfebe0f3c  55aa55aa 55aa55aa
)xfebe0f44  febff000 febe07dc
)xfebe0f4c  febe0fdc 000400db
```

Figure 12 Stack content after executing the attack, the return address is 0xfebff000.



```
      .offset 0x0090
      #if (MIP_MDP_ENABLE > 0x00000000)
        .extern _MIP_MDP
        jr _MIP_MDP
    0x90  .intvect..C.3A.5CCES+0x90:        07805b84         jr        MIP_MDP (0x5c14)
    0x94  .intvect..C.3A.5CCES+0x94:        .byte   00,00,00,00
    0x98  .intvect..C.3A.5CCES+0x98:        .byte   00,00,00,00
    0x9c  .intvect..C.3A.5CCES+0x9c:        .byte   00,00,00,00
```

Figure 13 MPU exception is triggered due to violation of execution rights in RAM.



```
    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 14 Malicious routine is successfully entered after the stack overflow.

the normal security checks that are required to enter flash programming mode. Due to the 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 13.

Note the MPU exception is non-maskable which is important to prevent an attacker from having the ability to delay or block the exception. Once the exception is triggered, the system can log the address where the violation occurred and store that in non-volatile memory for intrusion analysis. The next action is to reset the system which restores the CPU back to its original state. 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. Note the above action has to take into consideration the vehicle state to avoid creating a sudden loss of a safety function which would violate a safety goal.

10 Conclusion

In this paper, we presented methods for increasing the security resilience of AUTOSAR based systems. First by considering safety mechanisms as potential attack vectors, we offered several countermeasures that can prevent the abuse of safety mechanisms by attackers. In cases where full prevention was not possible, we proposed an enhanced fail-safe response that factors in malicious attacks as the potential source of the safety failure. Then we analyzed AUTOSAR safety mechanisms that exhibit security properties. We showed that AUTOSAR offered several strong security features if they are used with the proper constraints. We also showed the need to harmonize the response to fault detection to satisfy both the safety and security objectives of the system. In future work we plan on performing our evaluation on a commercially available AUTOSAR stack to produce a set of security requirements that can be integrated with AUTOSAR. We also plan to study the impact of physical attacks on the hardware protections such as RAM and Flash Test features.

References

[1] RH850 P1X Microcontroller Information microcontroller description. Available at: https://tinyurl.com/ybqbbanb [Accessed: 2017-11-28].

[2] Specification of Core Test. AUTOSAR Release 4.2.2

[3] Specification of Crypto Service Manager. AUTOSAR Release 4.2.2

[4] Specification of Diagnostic Event Manager. AUTOSAR Release 4.2.2

[5] Specification of Flash Test. AUTOSAR Release 4.2.2

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

[7] Specification of Operating System. AUTOSAR Release 4.2.2

[8] Specification of RAM Test. AUTOSAR Release 4.2.2

[9] Specification of Watchdog Manager. AUTOSAR Release 4.2.2

[10] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. (2009). Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13, 4.

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

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

[13] Cho, K.T., and Shin, K.G. (2016). "Error handling of in-vehicle networks makes them vulnerable," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security,* ACM, 1044–1055.

[14] Day, D.J., and Zhao, Z.X. (2011). 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, 472–483.

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

[16] Foster, J.C., Osipov, V., Bhalla, N., and Heinen, N. (2005). *Buffer Overflow Attacks: Detect, Exploit, Prevent*. Syngress Publishing (2005).

[17] Francillon, A., and Castelluccia, C. (2008). "Code injection attacks on harvard-architecture devices," in *Proceedings of The 15th ACM Conference on Computer and Communications Security*, ACM, 15–26.

[18] Francillon, A., Perito, D., and Castelluccia, C. (2009). "Defending embedded systems against control flow attacks," in *Proceedings of The First ACM Workshop on Secure Execution of Untrusted Code*, ACM, 19–26.

[19] Fürst, S., and Spokesperson, A. (2015). AUTOSAR the next generation – the adaptive platform. CARS@EDCC2015.

[20] GbR, A.: Specification of sw-c end-to-end communication protection library.

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

[22] Hartwich, F. (2012). Can with flexible data-rate. *Proc. iCC. Citeseer* (2012).

[23] Lima, A., Rocha, F., Völp, M., and Esteves-Verissimo, P. (2016). "Towards safe and secure autonomous and cooperative vehicle ecosystems," in *Proceedings of the 2nd ACM Workshop on Cyber-Physical Systems Security and Privacy*, ACM, 59–70.

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

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

[26] Standard, I.: Iso 26262, 2011. Road vehicles Functional Safety (2011).

[27] Standard, I.: Iso 11898, 2015. Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signaling (2015).

[28] Tencent: New car hacking research: 2017, remote attack tesla motors again. Keen Security Lab Blog. Available at: https://tinyurl.com/yalxvnoz

[29] Trapp, M., Adler, R., Förster, M., and Junger, J. (2007). Runtime adaptation in safety-critical automotive systems. *Software Engineering*, 1–8.

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

Biographies

Ahmad MK Nasser is a Ph.D. candidate at the University of Michigan Dearborn. He attended Wayne State University where he received his B.Sc. in Electrical Engineering and M.Sc. in Computer Engineering. Ahmad has held various Software Engineering roles throughout his career since 2002 with a focus on basic embedded software and embedded vehicle security. He is a domain expert in embedded systems, flash programming, vehicle diagnostics, communication protocols, AUTOSAR basic software, and hardware based security. He currently works as a senior software manager at Renesas Electronics America, where he leads the secure software center of competence. Ahmad is currently completing a doctorate in Computer Science at the University of Michigan Dearborn. His Ph.D. work centers on the interplay of safety and security in Automotive Systems.

Di Ma is an Associate Professor in the Computer and Information Science Department at the University of Michigan-Dearborn. She also serves as the director of the Security and Forensics Research Lab (SAFE). She is broadly interested in the general area of security, privacy, and applied cryptography. Her research spans a wide range of topics, including smartphone and mobile device security, RFID and sensor security, vehicular network and vehicle security, computation over authenticated/encrypted data, fine-grained access control, secure storage systems, and so on. Her research is supported by NSF, NHTSA, AFOSR, Intel, Ford, and Research in Motion. She received the Ph.D. degree from the University of California, Irvine, in 2009. She was with IBM Almaden Research Center in 2008 and the Institute for Infocomm Research, Singapore in 2000–2005. She won the Distinguished Research Award of the College of Engineering and Computer Science in 2017 and the Tan Kah Kee Young Inventor Award in 2004.

Priya Muralidharan has a Bachelors in Physics and Masters in Information Technology from the University of Delhi, India. She also has a Masters in Computer Science from the University at Buffalo, New York. She is currently working as a Senior Application Engineer at Renesas Electronics America, in the area of Functional Safety. She has over 10 years of experience in embedded software and controls development for various automotive applications such as Electric Power Steering Systems, Hybrid and Electric Vehicles. She has also worked extensively on electronic components such as electric and oil pumps as well as vehicle gateways.