Attacking AUTOSAR using Software and Hardware Attacks

Pascal Nasahl
Graz University of Technology
Graz, Austria
pascal.nasahl@iaik.tugraz.at

Niek Timmers
Riscure - Security Lab
Delft, The Netherlands
timmers@riscure.com

***Abstract*— The AUTomotive Open System ARchitecture (AUTOSAR) development partnership is a world-wide initiative aiming to jointly develop and establish an open industry standard for automotive E/E software architectures. This standard is rapidly being adopted by the automotive industry and therefore it is important to understand the attack surface of AUTOSAR-based electronic control units (ECU). In this paper we describe several scenarios how software and hardware attacks can compromise the security of AUTOSAR-based ECUs. We consider an attacker with physical access to the ECU who is capable of exploiting both software and hardware vulnerabilities. We discuss how an attacker can use different attack techniques to exploit these vulnerabilities. Moreover, we describe a case study in full detail where we execute arbitrary code on an AUTOSAR-based demonstration ECU by performing a voltage fault injection attack on the AUTOSAR communication stack. Several automotive threats may materialize if an attacker is able to execute arbitrary code on an ECU. For example, it will be possible to persistently modify the ECU's functionality if its code is not authenticated using secure boot.**

I. INTRODUCTION

In the past 30 years, the electrification of cars made a huge leap. Whereas in the beginning only simple control logic had been deployed in a car, demanding tasks like powertrain control and signal processing for autonomous driving are nowadays handled by computers. A modern car nowadays depends on tens of different computers, typically referred to as Electronic Control Units (ECU), ranging from sensors to high-performance computing platforms. The automotive industry quickly realized that some form of standardization is needed as multiple suppliers and manufactures are working together to produce a single modern car. Starting with OSEK/VDX [14] in the 90s, the AUTOSAR standard evolved slowly. The goal of this standard, which is developed by the AUTOSAR consortium, is to provide a common software architecture specification for all manufactures and suppliers who are part of the automotive industry [1].

The threat model for AUTOSAR-based ECUs should include local and remote attackers, attackers with significantly different budgets, different skills and completely different motives. The AUTOSAR standard and the standards on which it builds (e.g. *MISRA C*), focus mostly on robust software which is fundamental for security and safety. Nonetheless, it is not unlikely ECUs will be attacked using hardware attacks.

In this paper we provide an introduction to AUTOSAR which serves as context for the remaining sections. An attacker may identify software vulnerabilities by reviewing code (if source code is available) or by reverse engineering (if only binary software is available). During our research we did not have a real AUTOSAR-based ECU software available. We used a freely available AUTOSAR software stack [22] in order to create a demonstration platform. We describe several scenarios how software vulnerabilities may be introduced into an AUTOSAR-based ECU.

An attacker may resort to other types of attacks when software vulnerabilities are unknown. We describe several of these attacks, which range from simple PCB-level attacks (e.g. debug interfaces) to more advanced hardware attacks like fault injection. The results of these hardware attacks may end up in control of the ECU and/or extraction of (secured) information. Additionally, if no proper code signing mechanism (i.e. secure boot) is implemented, it may also lead to modification of the ECU's firmware. This is not an unlikely scenario as it is often believed secure boot is not required for devices that store and execute code only from internal memories.

The introduction to AUTOSAR is described in Section II. In Section III we describe the Software Attacks on AUTOSAR and in Section IV we describe the Hardware Attacks on AUTOSAR. The case study where we perform a fault injection attack is described in Section V. Finally, in Section VII we provide an overall conclusion.

II. AUTOSAR

In an automotive environment, the tasks and responsibilities of an operating system differ from classic computer systems. In such environments, the focus is on real-time processing of sensor information and heavily using bus protocols for communication with other ECUs. The AUTOSAR specification uses a three layer abstraction approach to simplify software development. Microcontroller specific drivers, the kernel, as well as the communication stack are implemented in the basic software layer (BSW). User applications can be developed independent of the underlying hardware by using the runtime environment (RTE) which provides access to features of the BSW. This layered approach is depicted in Figure 1.

The AUTOSAR standard is available in two significant different flavors: the AUTOSAR Classic platform and the AUTOSAR Adaptive platform. The AUTOSAR Classic platform is statically linked and not very flexible. It is often used for core ECUs deep within the car. The AUTOSAR Adaptive platform, in contrast, is dynamically linked, utilizes C++14 features and is therefore very flexible. It is often used for high performance and heavily connected ECUs. It is unknown at the moment of writing what the global market

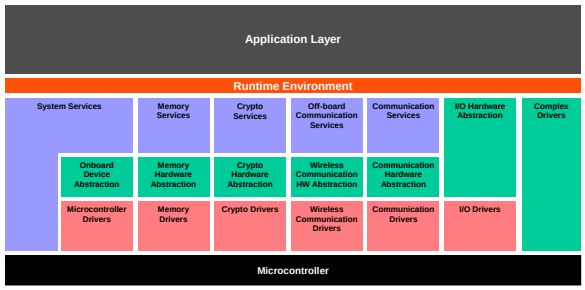

Fig. 1. AUTOSAR layered software architecture [2].

share of these platforms within modern cars is. Nonetheless, the AUTOSAR consortium includes all major vendors in the automotive market. Therefore, we expect AUTOSAR to have a significant presence in modern cars. Still, in this paper we focus primarily on the AUTOSAR Classic platform.

III. Software attacks

The AUTOSAR standard aims at producing safe and secure software for ECUs. It builds on top of well-accepted safe and secure coding standards like *MISRA C* and *CERT C*. An AUTOSAR-based software stack that fully adheres to these two coding standards should be safe and secure. In this section, we provide several scenarios where software vulnerabilities may be introduced into different AUTOSAR components. Note, that the examples provided in this section are by no means an exhaustive list.

A. MCAL vulnerabilities

The Microcontroller Abstraction Layer (MCAL) is often bundled together with the MCU. This means that often the MCAL is developed by the MCU manufacturer. Therefore, the MCU's MCAL needs to conform with the AUTOSAR specification if an ECU manufacturer wants to integrate it. All major MCU manufacturers provide such AUTOSAR compliant MCAL packages for their MCUs. If the MCAL does not adhere to the AUTOSAR standard properly, it may introduce software vulnerabilities when the MCAL is integrated by the ECU manufacturer. For this reason, it is essential that the developer of the MCAL provides assurances that the MCAL is implemented securely. Additionally, the integrator of the MCAL may conduct a (security) code review to check if the MCAL is implemented securely according to AUTOSAR's requirements.

It often depends on how the MCAL's functionality is used by the upper layers of the AUTOSAR software stack. For example, if there is an exploitable integer overflow vulnerability present in the EEPROM driver, it may not be exploitable if there are restrictions on the arguments that can be passed to this vulnerable EEPROM driver. Nonetheless, we believe it is important to make sure that vulnerabilities are mitigated at all layers of the AUTOSAR software architecture. Especially considering the usage of the vulnerable driver may change in future products rendering the software vulnerability exploitable. In other words, the MCAL should not trust the data passed on by the upper layers blindly.

B. Communication services vulnerabilities

The introduction of new and complex communication stacks (e.g. Ethernet) impacts the automotive industry. They require parsing of complex data structures whereas traditional communication stacks (i.e. CAN) do not. Especially the protocols that will be build on top (i.e. TCP/IP, UDS, etc.) are prone to software vulnerabilities. Proven communication stacks originating from general purpose computing (e.g. Linux) can often not be used by RTOS-based ECUs. They simply do not adhere to the requirements of a RTOS. The communication stacks that do qualify, however, are often not properly tested. Recent research conducted by Ori Karliner identified that the TCP/IP stack of FreeRTOS was riddled with exploitable vulnerabilities [15]. This vulnerable TCP/IP stack was also used by SAFERTOS, which is pre-certified to ISO 26262 ASIL-D. Communication stacks should be reviewed properly and the integrators of these communication stacks need to start requiring assurances that such a review is properly conducted.

IV. Hardware attacks

Hardware attacks require the attacker to be physically present to the target. Most hardware attacks, except when interfaces are used that are exposed externally, require the attacker to open the device. Several of the attacks described in this section require specialized tooling, some of which is nowadays easily available. Other high-end tooling may still be in reach for well funded actors like organized crime and state actors.

A. PCB-level

Printed Circuit Board (PCB) level attacks include basic hardware attacks where an attacker opens an ECU in order to probe and/or modify its PCB. Attackers often do this in order to extract the contents of non-volatile memory, access debug interfaces or sniff the communication between chips. Dependent on the ECU's design, other attacks may apply too. An ECU may be fully compromised if no sufficient measures are taken in order to prevent these attacks. If the ECU cannot be fully compromised, the information obtained from PCB-level attacks is often used to perform more advanced attacks like fault injection and side-channel analysis.

1) Modification of external memory contents: Attackers may use a standard flash programmer in order to modify the contents of external memories (e.g. eMMC). An attacker who is able to modify code stored in external memories will be able to execute arbitrary code on the ECU. Indirectly, this may also be achieved by modifying data which results in an exploitable software vulnerability. Code and data that originate from external memories should not be trusted. Therefore, any code and data that is stored in external memories should be authenticated before usage (i.e. secure boot should be implemented).

2) Extraction of external memory contents: Attackers may use a standard flash programmer in order to extract the contents of external memories (e.g. EEPROM). If any security sensitive data is stored unprotected (i.e. unencrypted) in external memory, this may be compromised by an attacker with physical access to the device. Therefore, it is important that the security sensitive data which is stored on external memory chips is encrypted.

3) Debug interfaces: All MCUs have a hardware debug interface that can be used to access the MCU's internals. Therefore, these debug interfaces should be properly protected. The tooling [17] and knowledge [18] is affordably available to communicate with hardware debug interfaces. An attacker who is able to communicate with a hardware debug interface is also able to compromise the security of an ECU entirely. Most OEMs demand these debug interfaces to be protected. Due to this reason, this threat is often properly mitigated.

4) Signal modification: Security critical data should not be stored in external memory, especially if the integrity of this data needs to be guaranteed. An example where this can go wrong is the try counter of the SecurityAccess service part of the Unified Diagnostic Services (UDS), which should have been implemented as-is according to the AUTOSAR standard. A ten minute try delay is triggered after three incorrect authentication tries are performed. If the try counter is stored in external flash, an attacker can prevent the try counter from being written correctly by modifying the external flash signals on the PCB. This attack is described in more detail by the authors of [16].

B. Fault injection

Fault injection attacks alter the intended behavior of a chip by affecting its environmental conditions. Glitches are injected into the target in order to affect the intended behavior of hardware and software [4]. Basic fault injection techniques affect the chip's clock and/or voltage [6] supply whereas more advanced fault injection techniques inject electromagnetic pulses and/or laser pulses [6] into the chip's surface [5][6]. The tooling required for these types of attacks are nowadays available to the masses for an affordable price [27].

The type of faults introduced into a target by these glitches (i.e. the fault model [8]) is often studied by academia. The common denominator between the academic research is that it is possible to modify instructions while they are being executed which is a strong attack primitive. Listing 1 shows an example of an instruction corruption for the *ADD* instruction on an ARM AArch32 system. Only by flipping a single bit in the instruction code, the result of the addition can be changed. This impacts the intended behavior of software completely. Therefore, fault injection attackers completely undermine the software security model most ECUs rely on.

```
add r0,r1,r2: 1110 1011 0000 0001 0000 0000 0000 0010
add r0,r1,r3: 1110 1011 0000 0001 0000 0000 0000 0011
```

Listing 1. Corruption of *ADD* instruction.

Traditional fault injection attacks aim to alter the intended behavior of a chip when a security critical decision is made [19], whereas more advanced attacks aim to take control of the chip by executing arbitrary code [20]. The case study described in Section V demonstrates such an advanced attack.

C. Side-channel analysis

Side-channel analysis (SCA) attacks are passive attacks where a side-channel is used in order to obtain information from a device. Common side-channels are timing, power consumption and electromagnetic emissions. These attacks are often used to extract secret keys from the device or bypass an authentication mechanism. Typical targets used by the automotive industry are the UDS SecurityAccess check and cryptographic operations (e.g. HSM). These attacks work both on software and hardware implementations. More information about timing side-channels can be found in [23], whereas more information about power consumption based side-channels is available in [21].

V. CASE STUDY: FI ON AUTOSAR

In this section we describe an advanced fault injection attack where we take control of an AUTOSAR-based ECU. The ECU is implemented using a STM32F4 based development board, which is running the *Arctic Core for AUTOSAR v3.1*. Although we are aware of the fact that more recent versions of AUTOSAR are available, the communication stack that we target is implemented almost identical in newer versions of AUTOSAR. In addition, we assume an attacker who has physical access to the ECU and access to the ECU's firmware. The authors of [12] describe how firmware can be extracted from ECUs efficiently.

A. Fault model: instruction corruption

In our attack we rely on the fact that we can modify instructions by injecting glitches into the voltage supply of the MCU. This attack is not unique to the target MCU or the fault injection technique we use. Similar to the example shown in Listing 1, Listing 2 depicts the impact of changing a bit in an instruction due to a glitch. Instead of modifying an *ADD* instruction, we are modifying a *LDR* instruction to load a register value into *PC*. This allows us to hijack the control-flow of the process at the moment of the glitch. Load and store instructions are used to copy memory. For ARM chips, this is done using the *LDR* instruction or the more efficient *LDM* instruction where multiple 4-byte words are copied using a single instruction. From an attacker's perspective, the *LDM* instruction is more interesting as destination registers are configured using a single bit, meaning that only a single bit needs to be set from 0 to 1 in order to load a value from a register into *PC*. This attack, which was proposed by Timmers et al., only works as-is for the ARM AArch32 architecture [11]. In their work, they provided a detailed comparison of corrupting *LDR* and *LDM* instructions and showed that getting control of the program counter register is feasible [11]. Most other architectures do not allow loading a value into the

program counter register directly. Nonetheless, variants of this attack are applicable to other architectures as well.

```
ldr  r1 ,[ r2 ,#4]:  1111 1000 1101 0010 0001 0000 0000 0100
ldr  pc ,[ r2 ,#4]:  1111 1000 1101 0010 1111 0000 0000 0100
```

Listing 2. Corruption of *LDR* instruction.

When developing a fault model, one has to consider the characteristics of the target. Microcontrollers used in an automotive context are usually certified with an ASIL protection level in order to meet functional safety requirements [9]. However, glitches still allow an adversary to perform attacks on MCUs certified with the highest protection level ASIL-D [10].

B. Fault target: communication stack

In a car, sensor information, user input as well as commands from different ECUs are usually exchanged using a multitude of system buses (e.g. CAN, FlexRay, LIN, etc.). Therefore, the communication stack is a crucial element of each RTOS running on an ECU. To satisfy the requirements for most common on-board system buses, the AUTOSAR communication stack supports various bus protocols like CAN, Ethernet, FlexRay and LIN [3].

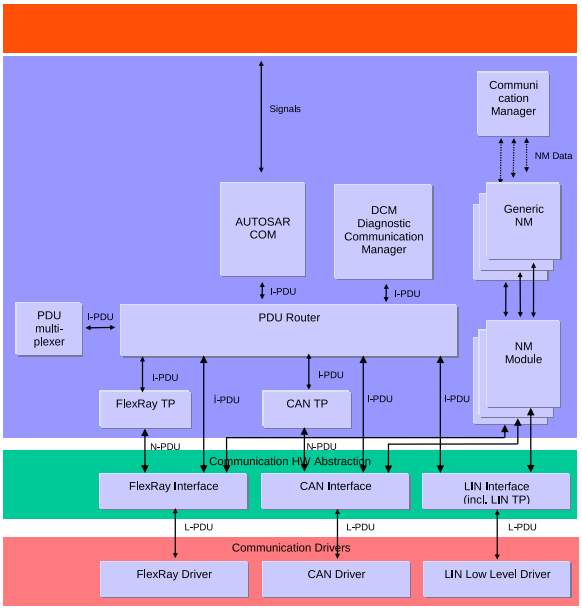

Fig. 2. The AUTOSAR communication stack [3].

Figure 2 depicts the current AUTOSAR communication stack. First, a bus interrupt is detected by the MCAL and the PDU is then forwarded to the corresponding communication driver. For CAN, either a CAN ISO-TP or a raw CAN frame can be received and handled. To bypass the length limitation of 8 data bytes for CAN, the ISO-TP protocol [13] is used. The protocol still makes use of 8-byte frames. When data up-to 4095 bytes is sent, the 8-byte frames are assembled on the ECU by the ISO-TP driver. The PDU router is responsible for routing the PDU to e.g. a user application or the diagnostic communication manager.

Our attack aims to load controlled data into the *PC* register of the processor. Therefore, we need to attack an interface that allows us to provide our own data. We assume this data is handled using *LDR* instructions. Timing the glitch in order to hit a single load instruction is difficult. Luckily, during a *memcpy* operation, multiple load and store instructions are executed consecutively in a loop. This makes the attack less dependent on timing. It does not matter which load instruction we modify successfully, as long as we modify one.

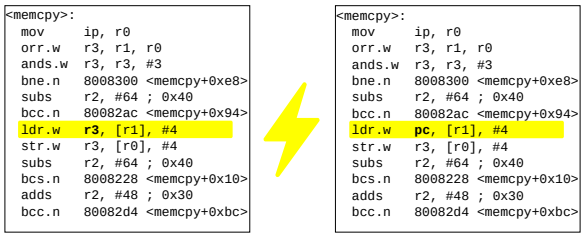

Fig. 3. Glitch injection moment during the *memcpy* function.

In Figure 3, we show a disassembled representation of the *memcpy* function used by the AUTOSAR communication stack. The data is copied from one buffer to the other using *LDR* and *STR* instructions in a loop. This function is used whenever data is sent to the ECU via the CAN bus (or any other bus). This data originating from the ECU is attacker controlled. It is actually copied multiple times by different AUTOSAR layers using the same *memcpy* function. This process provides an attacker with multiple opportunities to inject a successful glitch in order to load an arbitrary value into the *PC* register. We verified our hypothetical attack by modifying the communication stack's code. We modified one load instruction in order to load value originating from the CAN bus into the *PC* register. Our hypothesis was found to be correct: If we can modify a *LDR* instruction, we can also load an arbitrary value into *PC*. This finding provides us with enough confidence to move on to demonstrating a real attack whose results are described in Section V-C.

C. Attacking AUTOSAR

In this hardware attack on AUTOSAR, we are using voltage fault injection to gain arbitrary code execution on the control unit. To be independent of the user software implemented in the application layer, we are targeting AUTOSAR's communication stack. Using this attack, we are able to inject our own code as an independent task in the application layer. The advantage of controlling a task on the ECU is that we are able to add functionality that can be used maliciously. For example, we can add functionality to extract secrets from the ECU or perform subsequent attack relying on some form of control. An example would be a side-channel analysis attack to extract keys from the Hardware Security Module (HSM).

As stated in Section V-B, AUTOSAR supports sending up to 4095 bytes through CAN by using the ISO-TP protocol. Since the CAN bus does not support authentication by default, any participant with physical access to the bus is able to send

and receive data. For our attack we assume the attacker is able to remove the ECU from a donor car. By analyzing the source code of the AUTOSAR communication stack, we discovered that the PDUs are transmitted from the PDU router to the AUTOSAR COM or DCM module by using the *memcpy* function. An attacker will likely reverse engineer the firmware in order to form the same conclusion. By sending an address over the bus and then glitching *memcpy*, we can redirect the control-flow to an arbitrary position. We assume this attack may be applicable to most ECUs as copying data is such a fundamental operation.

AUTOSAR is a static real-time operating system, meaning that it is not possible to simply create a new task during runtime. The operating system manages tasks by using a task control block (TCB). In this entry, task name, priority, stack pointer and further important properties of the task are stored. The OSEK/VDX specification, which is the base of the AUTOSAR kernel, requires an OS IDLE task [14]. This task gets scheduled when no other task requests CPU time. For most ECUs this OS IDLE task will be implemented as an infinite loop. In other words, the AUTOSAR-based ECUs do not depend on the functionality provided by the OS IDLE task. This allows us to abuse the OS IDLE task in order to inject our own code. We can achieve this by simply overwriting the OS IDLE task pointer, which is stored in memory. This pointer is used whenever the OS IDLE task is scheduled. Note that targeting any other task is also feasible, but using the IDLE task has the advantage that the overall functionality of the system is not affected.

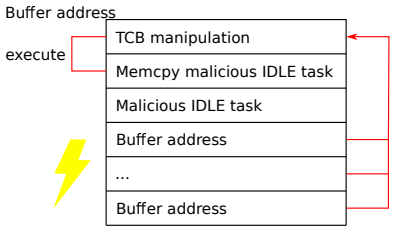

Fig. 4. Payload sent to the ECU via the CAN bus.

Figure 4 shows the payload sent to the input buffer of the ECU using the ISO-TP protocol. The payload consists of three main parts:

1) **TCB manipulation:** This code overwrites the stored task pointer on the stack of the IDLE task. The value of the new address is pointing to a region in SRAM, where the malicious IDLE task is placed.
2) **Copy malicious IDLE task:** The code of the new task is copied to the address selected before using *memcpy*.
3) **Buffer address:** The starting address of the input buffer is repeated several times in order to make the glitch less time dependent.

To place the malicious IDLE task to memory and to modify the task pointer of the IDLE task, the control-flow of the program is tampered by using fault-injection. As the input buffer is marked executable, the goal is to load the program counter register with the address of the input buffer. To reduce the complexity of the fault injection attack, the address of the input buffer is repeated several times. The attacker now only has to hit one of these addresses with the induced glitch. When the attack was successful, the next time the IDLE task will be scheduled, the code of the malicious IDLE task is executed.

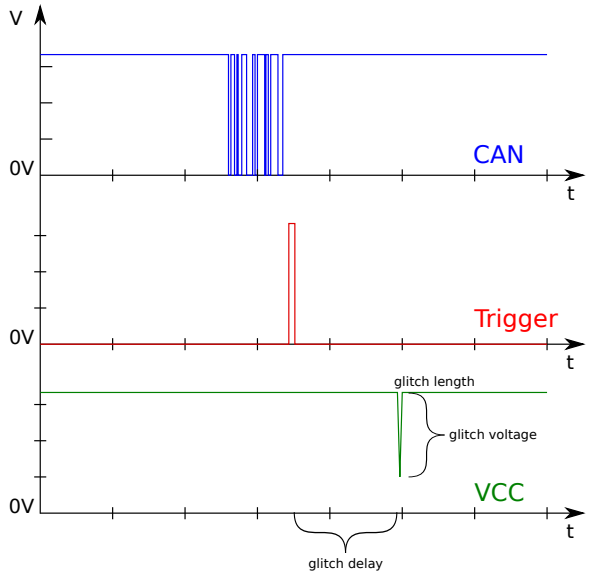

Fig. 5. Moment in time when the glitch is injected.

Figure 5 depicts the timing of our attack. First, the payload is transmitted using the CAN bus. Then, after the last frame has been sent, a trigger activates our glitch generator. As finding the required glitch strength for a successful glitch is complex [24], we are randomly modifying the glitch parameters: glitch length, voltage and delay. If the glitch injection was successful, the malicious IDLE task sends a message to the computer using UART. With our setup, we were able to gain arbitrary code execution on the ECU for around 2 times an hour. As Timmers et al. stated in their work, the success rate highly depends on the used setup and the device under attack and requires dozens of glitch attempts to be successful [11].

VI. HARDENING ECUS

An insecure design of an AUTOSAR-based ECU may allow an attacker to compromise its security using a logical vulnerability. The design is insecure when a vulnerability can be exploited by using the ECU's functionality according to the design. An example is a small key size for the ECU's authentication mechanism. If the design is strong, an attacker relies on a software and/or hardware vulnerability in order to compromise the ECU's security. These vulnerabilities are often introduced by developers when the design is translated into an implementation. Using software and/or hardware attacks, these vulnerabilities may be exploited in order to compromise the ECU's security. It is often agreed that translating a design into an implementation cannot be done without introducing

vulnerabilities. Therefore, it is important to make it as hard as possible for an attacker to exploit these vulnerabilities.

A. Making software attacks hard

Secure embedded devices, like smartphones, in order to be secure, rely on software exploitation mitigation techniques which originated from the general purpose world. For example, on modern smartphones, it is standard to implement stack buffer overflow protection (e.g. stack cookies), stack/heap memory protection (e.g. W⊕X) and address space layout randomization (e.g. ASLR). Implementing these software exploitation mitigation techniques will increase the complexity of exploiting a memory corruption based software vulnerability significantly. Unfortunately, the support for these basic exploitation mitigation techniques is non-existent or limited [25] in modern ECUs.

B. Making hardware attacks hard

Hardware attacks that rely on exploiting a hardware vulnerability, such as fault injection, are difficult to mitigate. These attacks are able to break the ECU's software security model on which it relies on. Most MCUs that are available for ECU designers are not hardened against hardware attacks like fault injection. Therefore, a designer of an ECU can only try to harden the ECU against hardware attacks using the MCU's standard functionality and the software that is executed. Even though hardware attacks are hard to protect against, they are often only a stepping stone towards more relevant attacks [26]. For example, hardware attacks are used to extract firmware in order to find software vulnerabilities that can be exploited remotely. Any secrets (i.e. keys) present in this firmware will be exposed as well. Therefore, the impact of hardware attacks can often be minimized by not storing any secrets in the firmware and making sure exploiting software vulnerabilities is difficult. In more detail, do not rely on security by obscurity. Most code protection features implemented by MCUs are vulnerable to fault injection attacks.

VII. CONCLUSION

The security of AUTOSAR can be compromised using software and hardware attacks. Although remote attacks are most critical, attacks that are performed locally still impact the security of modern cars. Local attacks are often a stepping stone for performing remote attacks (e.g. extraction of firmware) or simply sufficient for the attacker's use case (e.g. compromising assets). The impact of attacks where a physical presence is required by the attacker should be minimized. Security sensitive data should not be stored in the firmware.

In our case study, we showed that it is possible to take control (i.e. execute arbitrary code) of an AUTOSAR-based ECU using a fault injection attack. We injected the glitch in the *memcpy* function used by the AUTOSAR communication stack in order to cause a fault that loads an attacker controlled value into the *PC* register of the MCU. This allows us to hijack the MCU's control-flow in order to take full control of the ECU. Several automotive threats may materialize if an attacker is able to execute arbitrary code on an ECU. For example, it will be possible to persistently modify the ECU's functionality if its code is not authenticated using secure boot.

ACKNOWLEDGMENT

This work would not have been possible without the help of our great colleagues at Riscure, thank you! We also would like to thank Graz University of Technology, especially the Institute of Applied Information Processing and Communications, for their support.

REFERENCES

[1] Fürst, Simon, et al. "AUTOSAR–A Worldwide Standard is on the Road." 14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden. Vol. 62. 2009.
[2] AUTOSAR Consortium. AUTOSAR-Layered Software Architecture. AUTOSAR Consortium, Tech. Rep., 2017, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf
[3] AUTOSAR Consortium. Specification of CAN Transport Layer. AUTOSAR Consortium, Tech. Rep., 2017, Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_CANTransportLayer.pdf, 2017
[4] Timmers, Niek, and Cristofaro Mune. "Escalating privileges in Linux using voltage fault injection." Fault Diagnosis and Tolerance in Cryptography (FDTC), 2017 Workshop on. IEEE, 2017.
[5] Baumann, Robert C. "Radiation-induced soft errors in advanced semiconductor technologies." IEEE Transactions on Device and materials reliability 5.3 (2005): 305-316.
[6] Bar-El, Hagai, et al. "The sorcerer's apprentice guide to fault attacks." Proceedings of the IEEE 94.2 (2006): 370-382.
[7] Alimi, Nejmeddine, et al. "An RTOS-based Fault Injection Simulator for Embedded Processors." International Journal of Advanced Computer Science and Applications 8.5 (2017): 300-306.
[8] Piper, Thorsten, et al. "On the effective use of fault injection for the assessment of AUTOSAR safety mechanisms." Dependable Computing Conference (EDCC), 2015 Eleventh European. IEEE, 2015.
[9] Bellotti, M., and R. Mariani. "How future automotive functional safety requirements will impact microprocessors design." Microelectronics Reliability 50.9-11 (2010): 1320-1326.
[10] Wiersma, Nils, and Ramiro Pareja. "Safety!= Security: On the Resilience of ASIL-D Certified Microcontrollers against Fault Injection Attacks." Fault Diagnosis and Tolerance in Cryptography (FDTC), 2017 Workshop on. IEEE, 2017.
[11] Timmers, Niek, Albert Spruyt, and Marc Witteman. "Controlling PC on ARM using fault injection." Fault Diagnosis and Tolerance in Cryptography (FDTC), 2016 Workshop on. IEEE, 2016.
[12] Milburn, Alyssa, et al. "There Will Be Glitches: Extracting and Analyzing Automotive Firmware Efficiently."
[13] Zimmermann, Werner, and Ralf Schmidgall. Bussysteme in der Fahrzeugtechnik. Springer Fachmedien, 2006.
[14] OSEK, OSEK/VDX Operating System Specification 2.2.3, 2005, http://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf
[15] FreeRTOS TCP/IP Stack Vulnerabilities Put A Wide Range of Devices at Risk of Compromise: From Smart Homes to Critical Infrastructure Systems, 2018, https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-put-wide-range-devices-risk-compromise-smart-homes-critical-infrastructure-systems/
[16] Ramiro Pareja and Santiago Cordoba, "Fault injection on automotive diagnostic protocols", escar USA 2018
[17] Grand Idea Studio, http://www.grandideastudio.com/jtagulator/
[18] SENRIO, https://blog.senr.io/blog/jtag-explained
[19] Bypassing Secure Boot using Fault Injection, https://www.blackhat.com/docs/eu-16/materials/eu-16-Timmers-Bypassing-Secure-Boot-Using-Fault-Injection.pdf
[20] Viva la Vita Vida, https://media.ccc.de/v/35c3-9364-viva_la_vita_vida
[21] Introduction to Side-Channel Attacks, https://link.springer.com/chapter/10.1007/978-0-387-71829-3_2
[22] Arctic Core for Autosar v3.1, https://www.arccore.com/products/arctic-core/arctic-core-for-autosar-v31
[23] Timing attack, https://en.wikipedia.org/wiki/Timing_attack

[24] Carpi, Rafael Boix, et al. ”Glitch it if you can: parameter search strategies for successful fault injection.” International Conference on Smart Card Research and Advanced Applications. Springer, Cham, 2013.
[25] Dissecting QNX, Jos Wetzels, Ali Abbasi https://www.blackhat.com/docs/asia-18/asia-18-Wetzels_Abassi_dissecting_qnx__WP.pdf
[26] There Will Be Glitches:Extracting and Analyzing Automotive Firmware Efficiently, Alyssa Milburn et. al., https://www.riscure.com/uploads/2018/11/Riscure_Whitepaper_Analyzing_Automotive_Firmware.pdf
[27] NewAE Technology Inc., ChipWhisperer, https://newae.com/tools/chipwhisperer/