Fuzzy Fault Injection Attacks Against Secure Automotive Bootloaders

Enrico Pozzobon[1,2,3], Nils Weiss[1], Jürgen Mottok[3], and Václav Matoušek[2]

[1] Dissecto GmbH
[2] University of West Bohemia
[3] OTH Regensburg

Abstract. Secure embedded bootloaders are the trust anchors for modern vehicles' software. The secure software update process of ECUs is well-defined across the entire automotive industry. Every OEM has his own implementation, but follows the general software update process. This paper demonstrates code execution attacks by combining software and hardware weaknesses in secure automotive bootloaders. The attack can be performed entirely automated, no static code analysis is required. Random fault injection parameters were sufficient to obtain code execution in a reasonable time. All experiments were conducted with electromagnetic fault injection and without any hardware modifications of the targets. We successfully performed our attack on two entirely different gateway Electronic Control Units (ECUs) used in current vehicles (at the time of this research) from Volkswagen and BMW. As a result of this attack, consisting of a combination of a hardware and a software attack, the general secure software update process used in the automotive industry needs to be revised.

1 Introduction

Modern vehicles possess a unique threat landscape, distinct from that of other connected devices. On one hand, vehicles are susceptible to remote attacks similar to those faced by Internet of Things (IoT) devices or personal computers. Attacks in this threat category were mainly demonstrated by researchers [2,7,11,17,18]. Until now, remote attacks from malicious parties, excluding attacks against keyless entry systems, weren't observed by the public. We assume that this is due to the lack of a clear financial incentive. On the other hand, vehicles are also vulnerable to attacks from individuals with physical access to the system, such as in the case of car thefts or chip-tuning activities. These scenarios are particularly relevant in the real world, as evidenced by statistics on car thefts, and are driven by the existence of a market for stolen cars and components or chip-tuning software [3].

In order to defend against these various threat groups, the automotive industry employs a defense-in-depth strategy that incorporates security technology at various layers of the system, from backend security and secure vehicle network architectures to microcontrollers in ECUs. The hardware security of ECUs is particularly crucial for the success of this defense strategy, as hardware attacks are a common method for obtaining firmware for information gathering and exploit preparation in remote attack scenarios, and are used directly to manipulate firmware or unlock debug interfaces in scenarios with physical access. Additionally, the introduction of new functionalities such as Features-on-Demand (FoD) will further challenge the hardware security of vehicles, as they present new opportunities for malicious actors.

One popular form of attack against microcontrollers and Hardware Security Modules (HSMs) is the use of Fault Injection (FI) techniques, which have become increasingly accessible with the advent of inexpensive hardware setups. This paper focuses on a specific type of FI attack known as Wild Jungle Jumps, which involve the manipulation of program counters to achieve code execution at arbitrary memory addresses [6].

1.1 Safe and Secure Microcontrollers

In many modern vehicles, security trust anchors are built with safe and secure microcontrollers, such as the MPC56xx and MPC57xx series from NXP. These controllers use the PowerPC (PPC) Variable Length Encoding (VLE) instruction set and are specifically designed for body-control or gateway applications. Therefore, the manufacturer equipped them with a feature set rich of security functions and a dedicated embedded HSM core. A wide range of security functionality allows high protection of the internal flash memories and debug interfaces of these processors.

In body control applications, these processors store the cryptographic material for the immobilizer system of the vehicle. Acting as a gateway, these microcontrollers perform firewall functionalities for Controller Area Network (CAN) and FlexRay buses and act as a host controller of managed automotive Ethernet switches including software updates and configurations. Once a malicious actor can control the gateway ECU, any traffic in the vehicle network can be manipulated

1.2 Repair shop testers

Repair shops and repair shop testers are crucial entities for an Original Equipment Manufacturer (OEM)'s ecosystem. In repair shops, cars without over-the-air update functionalities receive regular software updates in fixed maintenance cycles or because of a recall action. Furthermore, repair shops need to be able to swap ECUs and perform teach-in processes. These functionalities are provided by diagnostic protocols, which are a major attack surface for ECUs [20]. Since high-speed Internet connectivity is still not guaranteed in all places, many repair shop testers are designed to operate offline. This requires the presence of all firmware update files and security access algorithms in the repair shop tester software. All these firmware files, security access algorithms, and repair shop tester software can be found on shady Internet forums, torrent, or download portals, and hardware clones of OEM-Testers are sold online. These tools are illegal to redistribute, but the required effort to obtain or buy cloned hard- and software is extremely low. This allows attackers to use official tools for information gathering or attacks.

1.3 Secure Software-Update Process of ECUs

A simplified software update process of non-volatile memories in automotive control units, based on ISO 14229-1:2020 [5, p. 374], is shown by fig. 1. ISO 14229-1:2020, also called Unified Diagnostic Services (UDS), is the standard protocol for software updates in automotive systems and is used by most OEMs and suppliers in the world. Depending on the vehicle manufacturer, some variations of this process are possible. Some OEMs require the transfer and verification of an Erase-Routine. For safety-critical reasons, often the code to erase memory isn't part of the bootloader and needs to be transferred separately. This Erase-Routine enables erase functionality of the non-volatile memory (e.g. flash memory), which is necessary to perform write operations afterward. Another variation lies in the data transfer. Depending on the ECU, the transferred data can optionally be a compressed or encrypted binary file. Most ECUs, implementing the ISO 14229 standard software update procedure, allows the writing of arbitrary binary code to the ECUs' non-volatile memory. A signature verification to enable execution of the transferred binary code is always performed after the write to non-volatile memory.

2 Related work

In the paper "BAM BAM!! On Reliability of Electromagnetic Fault Injection (EMFI) for in-situ Automotive ECU Attacks [14]", the author performs an EMFI attack targeting the Boot Assist Module (BAM) present in older versions of the Freescale/NXP PowerPC microcontrollers. More recent models of PowerPC Microcontroller Units (MCUs) from the same manufacturers make use of a Boot Assist Flash (BAF) module instead, which is re-writable and thus vulnerable flash code there can be patched, so the attack does not affect these newer controllers.

Wiersma and Pareja [22] demonstrated an attack against the Device Configuration (DCF) system of recent PowerPC MCUs next to an analysis of the resilience of MCUs for safety-critical applications against fault injection attacks.

Wouters et al. [23] demonstrated voltage glitching on internal bootloaders of microcontrollers used in immobilizer systems. Through their attack, they could obtain the internal firmware and identified several security flaws in the immobilizer systems of major car manufacturers such as Toyota, Kia, Hyundai, and Tesla.

Attacks against internal bootloaders of three different MCUs were demonstrated and summarized by Van den Herrewegen et al. [21]. The researchers performed static and dynamic analysis and documented the first multi-glitch attack on a real-world target.

Nasahl and Timmers used glitching attacks on an evaluation setup to obtain code execution on an AUTOSAR-based demonstration ECU [12]. By leveraging fault injection weaknesses in the ARM Instruction Set Architecture (ISA) they could corrupt a `memcpy` operation to perform a jump into writable RAM memory.

3 Contribution

In this paper, we present an EM fault injection attack which grants unauthenticated code execution on modern ECUs by making use of the large programmable flash memory that can be easily written using leaked repair shop tools. The attack requires a single glitch and little to no knowledge of the target, and was tested on multiple ECUs based on different PowerPC MCUs. We first show the attack on a production ECU which outputs stack traces on a serial logging interface, and later generalize the attack on other ECUs which make use of the same PowerPC core architecture without the need of any feedback channel.

4 Test Setup

A test setup was built to perform the fault injection tests on real-world target ECUs and on an ARM-based evaluation board. The chosen technique was EMFI because it does not require any hardware modification of the target, so an exploited target is visually indistinguishable from an unaltered ECU, which is desirable from the attacker's point of view.

In any fault injection method, several parameters can be altered for a fault, which constitutes the search space for the successful attack parameters. For finding the correct parameters, it is important to be able to automate the setup,

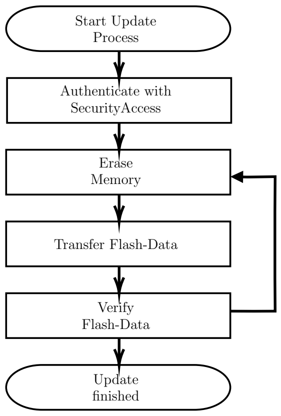

Fig. 1. Typical flow chart for a secure software-update procedure of non-volatile memory.

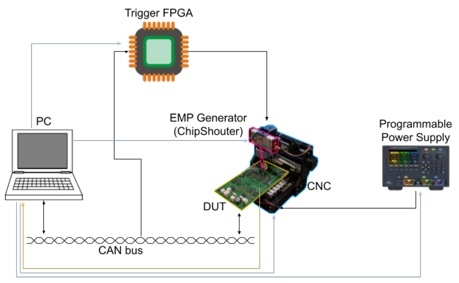

Fig. 2. Diagram of our automated test setup

so that the entire parameter search algorithm can be executed without human interaction. In an EMFI attack, the main parameters are the following:

– Injection **coil** (shape, size, number and direction of turns),
– **position** in space of the injection coil,
– **duration** of the activation of the coil,
– **voltage** across the coil,
– time **offset** from trigger signal (if the target firmware has deterministic execution time, this is equivalent to choosing which instruction to attack).

In our test setup, the setting of every one of these parameters could be automated, except for the injection coil, which must be changed manually. To reduce the manual work necessary, we only performed our tests with two coils included in the ChipShouter kit: a 1mm diameter core clockwise wound coil, and a 1mm diameter core counter-clockwise wound coil.

4.1 Description of the test setup

The hardware test setup for the collection of the data necessary for the attack is shown in fig. 2. It is composed of the following items:

– **USB-to-CAN** - for CAN communication with the target.
– **USB-to-UART** - for receiving debug logs from the target over a Universal Asynchronous Receiver-Transmitter (UART) connection.
– **ChipShouter** - for injection of the electromagnetic fault.
– **Computer Numerical Control (CNC) mill** - for manipulating the position where the electromagnetic fault is injected.
– **Field-Programmable Gate Array (FPGA)** development board - for consistently triggering the glitch on a specific CAN message.

– **Keysight E36313A power supply -** for power-cycling the ECU between attempts.

The target ECU is placed on the CNC mill bed and the CNC mill drill is replaced with an Electromagnetic Pulse (EMP) injection tip connected to a Chipshouter, which allows to place the injection tip in any position above the target MCU with a precision of $\pm 0.01\ mm$. The diagnostic CAN interface of the ECU is connected via a CAN bus to the control computer, and an FPGA is also connected to the same bus via a CAN transceiver to listen for a specific CAN frame and emit a delayed trigger signal when a match is found. This trigger signal is fed to the ChipShouter and allows for fine-tuning the timing of the glitch. The programmable power supply is used to power-cycle the ECU when necessary. Finally, a USB-to-UART adapter is used to collect feedback data from the target ECU.

The software used to control the setup was written in the Python programming language, using Scapy for the CAN and UDS communication [15]. A PostgreSQL database is used for logging and data analysis.

Exploit code as well as example code on the target was written in C, PPC and ARM assembly and compiled using the powerpc-eabivle-gcc and arm-none-eabi-gcc toolchains.

4.2 Target description

The initial target chosen for this attack was a gateway ECU from Volkswagen. The ECU makes use of an MPC5748G MCU, with a locked Joint Test Action Group (JTAG) debug interface. The target MPC5748G MCU is used in several ECUs by different manufacturers. It contains a total of 4 PowerPC cores: two e200z4 cores, one e200z2 low-power core, and one e200z0 HSM core. All cores run the PowerPC VLE instruction set. The UART logs emitted by the target ECU contain stack traces whenever an exception interrupt is called, including the values of all general-purpose registers and some special registers. While it is impossible to extract the firmware from the ECU over JTAG interface, UART interface, or UDS services, the application firmware for this ECU can be found in Open Diagnostic Data Exchange (ODX) flash container files leaked from repair-shop testers.

Leaked firmware files and UART logs were useful tools for studying the fault. Both are not necessary to perform a successful attack. Later on, the attack was tested successfully on different ECUs from other manufacturers, some of which did not have UART logging or leaked firmware files available.

Since the communication interface used by the repair shop hardware to flash the target ECU is CAN, the test setup was built so that the trigger for the glitch would be derived from a specific CAN frame. The glitch is triggered after the last ISOTP consecutive frame of a UDS request but before the corresponding response, as seen in fig. 3 [4]. This is done in an attempt to inject the glitch during the processing of the UDS request.

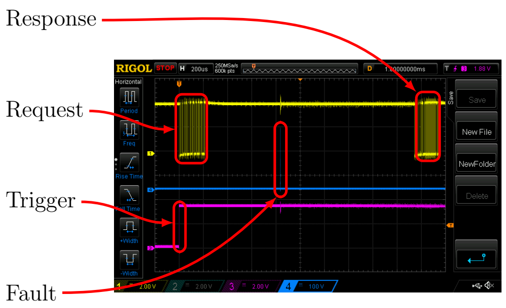

Fig. 3. A snapshot of the oscilloscope screen during a fault injection attack. The yellow line represents the CAN protocol, and shows the request and response ISOTP messages. The magenta line is the trigger from the FPGA, which detected the searched CAN frame. The blue line shows the voltage spike sent to the EMFI coil.

5 Information Gathering

This section describes our information gathering process and enhancements of information leakage by using fault injection attacks.

5.1 Stack-Traces and PPC exception handlers

The target MCU takes interrupts whenever an exception is generated, beginning the execution of the corresponding Interrupt Service Routine (ISR). Exceptions are generated by signals from internal and external peripherals, instructions, the internal timer facility, debug events, or error conditions. During development, exception interrupts can be used to diagnose programming errors (such as, a jump to an invalid instruction is detected) and run-time errors (such as an error happened when reading from the Error correction code (ECC) memory).

On the target ECU, ISRs associated to exception interrupts are programmed to output a stack trace over the UART interface and then resetting the MCU. The emitted stack traces contain all the general purpose registers, several special use registers, and the list of the addresses of the functions that are currently on the execution stack. An example can be seen in Listing 1.2. In particular, the special registers emitted include Save/restore register 0 (SRR0) and Critical Save/Restore Register 0 (CSRR0), which in general contain the address of the instruction that caused the interrupt. Machine Check Status Register (MCSR) and Exception Syndrome Register (ESR) are also emitted in the stack traces, which contain bit-masks detailing what kind of exception was generated to give information about the cause of the exception.

When a fault is injected, an exception may be generated and, if that happened, the corresponding interrupt will be taken, causing the processor to start executing the associated ISR. The values of ESR and MCSR can then be used to determine which exception was caused by the fault, while Link Register (LR), SRR0 and CSRR0 can be used to determine the address being executed by the processor at the time of the fault.

As illustrated from fig. 3, the fault was injected during the time interval between when a UDS request was sent and the response was received. In this situation, one of the following outcomes can happen whenever a fault is injected:

- Nothing anomalous happens and the correct UDS response is received.
- An undetected mistake is generated, causing a corrupted UDS response to be received and/or an unexpected message on the UART log.
- An exception is generated, and the processor emits a stack trace and the MCU resets.
- No stack trace is emitted and the MCU resets.

These stack traces found on real ECUs extracted from cars contain other information that could be useful for other attacks as well. For example, it is possible to trace the execution of the program over time by examining the value of LR (see Figure 5), or to reverse engineer which cryptographic algorithms are used by checking if magic numbers used by specific algorithms appear in the general purpose registers (for example, the initialization values of an hashing algorithm like in Figure 6).

5.2 Parameter search

The aim of the presented attack is to find a repeatable fault that causes the unsigned code stored in the flash to be executed, bypassing the signature check. The search space of all the possible faults corresponds to the multi-dimensional space (x, y, z, c, i, t, o) defined by the parameters described in section 4:

- (x, y, z): position of the coil in space
- c: Coil used
- i: intensity of the fault (current through the coil)
- t: duration of the fault
- o: time offset from trigger

A genetic algorithm was written [Reference removed to keep the paper anonymous] to search for fault parameters tuples (x, y, z, c, i, t, o). An initial population of parameter tuples is initialized with uniform random values, except for a parameter c which is fixed with every run of the algorithm since the coil type can not be automatically changed by the test setup. Each parameter tuple is tested by executing a fault on the automated setup; and a fitness score is assigned to the tuple depending on the effect of the fault. The score is then used to generate the next generation of the population by rewarding tuples that obtained exceptions that correlate positively with corruptions of the program counter.

The genetic algorithm was run initially after erasing the application firmware from the MCU and before flashing an exploit payload. Because of this, the best outcome of a fault was to cause an "Illegal Instruction" exception with an SRR0

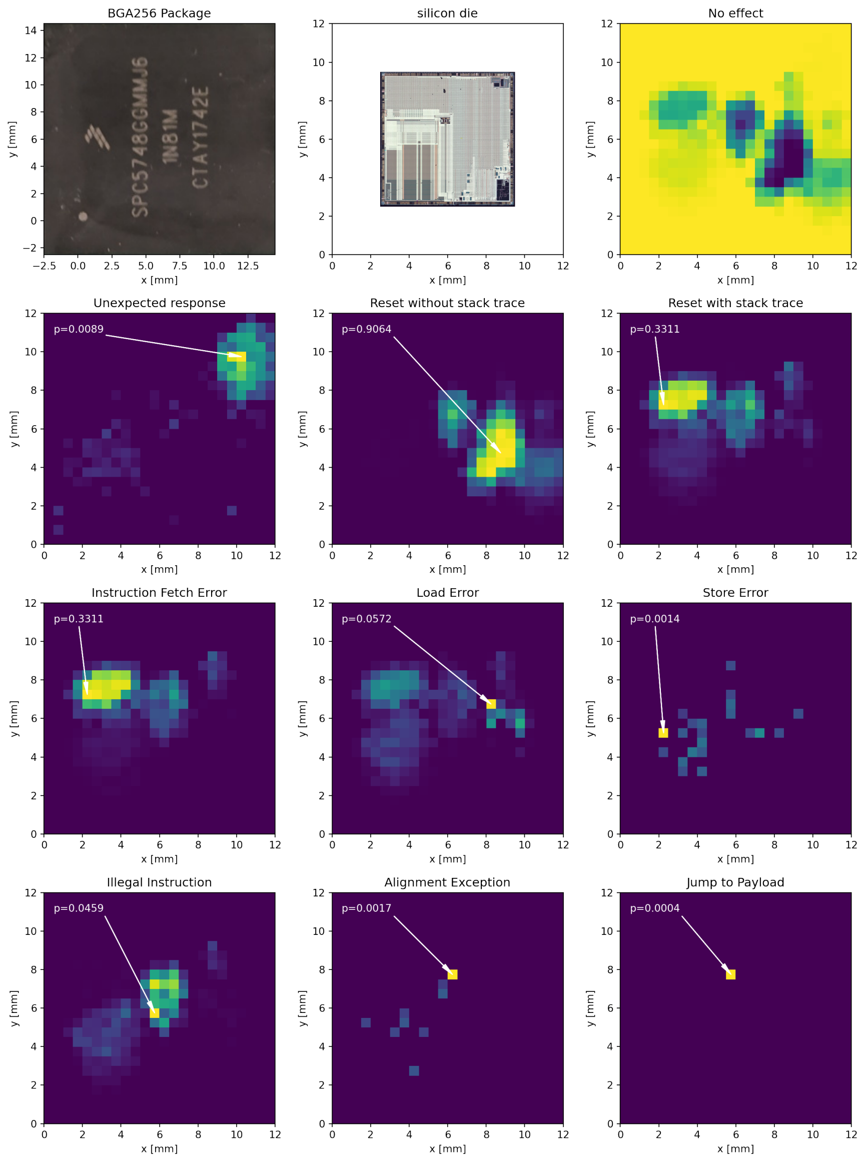

Fig. 4. Sensitivity of the different areas of the MPC5748G MCU package to the fault with respect to different errors. These images were drawn from unbiased data with random and uniformly distributed fault parameters, using the 1mm core diameter counter-clockwise wound coil. p indicates the probability of a fault for the $0.5\ mm \times 0.5\ mm$ area pointed by the white arrows, which is the highest probability in the relative image.

value within the erased memory, indicating that the MCU was attempting to execute code inside the erased memory reserved for the application firmware. This occurrence was rewarded with the highest fitness score in the genetic algorithm.

Later, to evaluate the effectiveness of the evolutionary algorithm in finding the optimal parameters for the glitch, the test was repeated with randomized fault parameters, which caused a success in only 9 out of 10256642 glitch attempts on the target. The unbiased attempts were also used to generate the data set for Figure 4, which shows the probability density of several faults occurring with respect to the x and y parameters. The first plot simply shows the MCU package for reference of the coordinates. The plots named ”No effect”, ”Unexpected response”, ”Reset without stack trace” and ”Reset with stack trace” correspond to the mutually exclusive outcomes described in section 5.1. The other plots represent different error states that can be inferred from the flags set in the registers leaked through the stack traces. In particular, ”Jump to Payload” indicates that a stack trace was emitted with SRR0 within the interval of the erased flash and having bit 36 set in the ESR [13].

Analyzing Figure 4, one can see that ”Jump to Payload” is highly correlated with ”Alignment Exception” and (in lesser measure) to ”Illegal Instruction”, because these exceptions are caused by faults that write the program counter. The ”Unexpected response” plot shows that corrupted CAN messages are often obtained when injecting faults in a specific area of the processor, which may indicate that the physical layer implementation of the CAN protocol is located in that area of the silicon. The ”Reset without stack trace” shows a sensitive area which has a 90% probability of causing an immediate hard reset when targeted by an electromagnetic fault, which may indicate the power supply circuitry is located there. ”Load”, ”Store” and ”Instruction Fetch” errors are correlated with targeting significantly different areas of the chip with faults, which also gives hints to where different parts of the Integrated Circuit (IC) are implemented in the silicon.

6 Vulnerability and Exploitation

After finding a set of fault parameters that are likely to lead to the corruption of the program counter in some way, it is necessary to direct the program counter to the desired unsigned code address in some way.

6.1 Directed jumps to memory

The method typically used to reach injected unsigned code is to attempt to load externally controlled data into the program counter, similar to the attack of Timmers et al. [19] on ARM-based processors.

For example, transferring the desired destination address into the target using a CAN frame, then injecting a glitch at the moment when that data is copied somewhere else, it is possible that the destination encoded in the assembly instruction is corrupted to become the program counter.

This kind of fault is harder to inject on a PowerPC target compared to an ARM target since loads into the program counter are not encoded in a similar way to loads into any other register, so the probability of flipping all the bits necessary to cause a load into program counter is extremely low.

Moreover, this kind of attack requires the attacker to know where his payload was transferred (which is not always the case) in order to send the exact destination address to the target memory. The payload is usually transferred into a large buffer that can be manipulated by the attacker (such as the ISOTP message buffer in the case of an automotive target), which means the payload needs to be transferred before every fault attempt, reducing the rate at which faults can be injected.

In practice, after 4 million injected glitches while trying different UDS messages as target buffers, no success was obtained on our PowerPC target using this technique.

6.2 Random jumps to application flash

After erasing the entire memory of the target using the official repair shop tester software, the entire flash memory of the MCU will contain the byte 0xff, with the exception of the small bootloader that will flash any received firmware and authenticate its signature before booting it. Since 0xffff is an invalid instruction in the PowerPC VLE instruction set, if the target MCU was to try executing this erased memory, it would throw an "illegal instruction exception", call the interrupt exception handler, and reset afterward.

In some ECUs, the interrupt exception handler emits a small stack trace whenever an exception happened, which included, among others, the address that was being executed when the exception happened as well as the type of exception. This allowed us to verify that, while injecting faults, the program counter was sometimes jumping to random locations in the erased application flash, attempting to execute it, and consequently emitting an illegal instruction exception stack trace on the UART interface. fig. 4 shows how exceptions reported on the UART stack traces correlated with the position of the fault injection coil over the target MCU.

6.3 Weak authentication for persistent memory writes

Extracted UDS security access algorithms from repair shop tester software became publicly available in open source projects, for example on GitHub [8, 10]. Therefore, we treat the security access authentication in the standardized firmware update process, shown in fig. 1, as widely broken. Even if a security access algorithm is not hosted on these open source projects, our attack can be performed by abusing the original repair shop tester to write our payload for our attack to an ECUs program memory, just by replacing the firmware update files in the directory of the repair shop tester software.

These tools allowed us to write custom firmware to the application flash of different real-world target ECUs. The written content can't be executed, since signature checks will fail. To apply our attack on ECUs from BMW, we used the software tool E-SYS. For ECUs from Volkswagen, we flashed ECUs by using the open-source software VW-Flash [9].

6.4 Exploit: Execution of arbitrary code

Finally, we can combine the described hardware and software vulnerabilities to obtain arbitrary code execution. First, a firmware is assembled where a small payload (with entry point named `start`) is preceded and followed with long ”trampoline” sections programmed with `NOP` slides interleaved with unconditional branches to the payload entry point. This firmware was flashed to the entire application flash memory of the target.

```
.rept 1000
.rept 113
        se_nop
.endr
        e_b _start
.endr
_start:
        // The actual exploit code is written here
```

Listing 1.1. Example GNU assembler code which generates a long PPC nopslide which interleaves one branch instruction every 113 NOP instructions.

Since PPC VLE instructions can be aligned at every even address, and since the branch instruction is 4 bytes long, if the fault causes a jump in the middle of a branch instruction, it would cause an illegal instruction exception. Since the `NOP` instruction is 2 bytes long, it is important to keep the ratio of branch instructions to `NOP` instructions very low to minimize the probability of this happening.

Similarly, if the fault causes a jump in the middle of the payload code, it would likely not function correctly since the initialization code of the payload would not have been executed. The probability of this happening can be reduced by keeping the payload size small.

Ideally, we want to inject the fault during the execution of a large UDS job that involves a variety of machine code to increase the probability of encountering an instruction that, when glitched, can lead to the corruption of the program counter. We tested all the available UDS jobs and choose the one that took the longest amount of time to execute, hoping it would be correlated with a large variety of instructions.

By applying random faults during the execution of a chosen UDS request, we cause random jumps to the application flash memory. Since the great majority of the target's memory contains our ”trampolines”, we have a high probability of jumping into one of them. Once the processor jumps there, it will reach our exploit code.

An advantage of using random corruptions of the program counter is that no large transfer of payload is necessary to prepare the target for the fault. After every failed fault, if the target did not reset, we can just re-send the UDS request and try again. In this way, we were able to test up to 2 faults per second. Another advantage is that it is not required to know where the payload will be placed exactly in the application flash, since it can be written as a position independent executable.

After using the genetic algorithm presented in [Reference removed to keep the paper anonymous], a fault that executes our unsigned payload was found on average after 950 injected faults, and the parameters found can reproduce the execution of the unsigned payload with a probability of 0.06. Without the search algorithm, our fault led to code execution around once every 10^6 attempts.

6.5 Impact: Looting secrets, unlocking JTAG

As a showcase of our attack, we wrote three different exploits. First, we simply printed a "Hello, World!" message on the UART interface to demonstrate the arbitrary code execution. Secondly, we wrote a payload that would dump the contents of the firmware over the UART interface. Finally, we wrote a payload that disabled the JTAG censoring permanently, allowing for more exploration and exploitation over the development interface.

Firmware extraction Crucial software components, such as the bootloader of an ECU, are not always part of repair shop tester firmware files. Also, special firmware files for security-critical ECUs, such as the immobilizer ECU, are often not part of available firmware leaks. To obtain these files, an attacker needs to extract the data from an ECUs flash memory. The presented hardware and software attack is suitable for this scenario, as it enables attackers to dump the firmware using a dumper payload to later study it in reverse engineering.

Extraction of secret data Modern cars have all kinds of secrets hidden in an ECUs firmware. Some examples are cryptographic keys for vehicle internal communication (AutoSAR SecOC [1]), cryptographic data for backend communication, shared keys inside bootloaders, secrets for vehicle immobilizers, and certificates for features on demand, just to mention some examples. All this data is valuable for attackers and need to be protected to guarantee a vehicle's security. Since our attack allows low-level code execution, any secret which is not protected from a dedicated HSM can be read out. Besides the fact that HSM co-processors are not affected by this attack, an attacker can control the HSM and is able to execute API calls.

Re-Enabling JTAG The MPC57xx microcontroller series allows locking the JTAG debug interface via so-called DCF records. These records are written in an One Time Programmable (OTP)-section of the internal flash memory where records can only be appended and never deleted. A low-level state machine (System Status and Control Module (SSCM)) is parsing all records sequentially during the power-up sequence of the microcontroller. For the censorship setting, which enables or disables the JTAG interface, the last appended record takes precedence, meaning it is possible to disable the censorship flag by appending a DCF record with a normal flash write.

Once we obtain code execution through a successful glitch, we can simply append a DCF record which re-enables the JTAG interface on these processors. We could successfully perform this attack by transferring the code example from the application note [16] to the application memory.

7 Generalization of the attack

The presented attack was successfully performed on three different ECUs. The only similarity between these ECUs was the MCU ISA and a secure bootloader following the ISO 14229 standard. Anything else, including the processor series, the firmware, the bootloader implementation, the hardware and software manufacturer, and even the OEM using the ECU are different. Furthermore, the attack was performed without any static analysis of the actual firmware on the target. Parameter search using genetic algorithm for the injected fault allowed us to perform a code execution attack within one hour on average. Since the similarities between our targets were marginal, we expect a wide variety of ECUs to be vulnerable to this attack. Additionally, we successfully demonstrated the application of the same algorithm on an ARM-based evaluation board for MCUs that are predicted to be used in the successors of the tested ECUs. It was found that the ARM processor architecture is significantly more susceptible to wild jungle jumps into writable memory areas, compared to PowerPC processors, though this might have been caused by the development board used being more susceptible to electromagnetic fault injection attacks due to the layout of the PCB compared to an automotive target.

As mentioned before, for targets that do not emit a stack trace upon encountering an exception interrupt, it is possible to ”train” most of the fault parameters on an off-the-shelf MCU with the same model as the attacked one, and then find the remaining ones via exhaustive search on the target itself (usually, this only involves finding the point in time to inject the fault upon, during the execution of a long UDS job).

8 Mitigation

The attack can be mitigated by using countermeasures against arbitrary code execution which are available on some hardware. The controllers found on the tested ECUs can temporarily limit the execution of code from the flash while it is not authenticated yet using the Memory Protection Unit (MPU) module, but this was not implemented by manufacturers. Particular attention must be placed on disabling the execution of the unauthenticated code early in the boot process, to minimize the attack surface for the presented attack. In general, the current software update process of secure firmware updates in automotive systems needs to be extended to prevent this or similar attacks.

We also speculate that the execution of the bootloader inside the HSM core or other types of secure elements can reduce the vulnerability of an ECU, since the code for these is execute from a section of the flash memory that is functionally separated from the large application flash. Moreover, the documentation for these cores is usually kept secret thus making the development of an exploit much harder.

9 Conclusions

We demonstrated the efficient application of fault injection attacks to obtain code execution through program counter manipulation on different real-world targets.

Thanks to commonly leaked "UDS Security Access" credentials, the attacker is able to inject a large amount of code in the program flash of the victim device, which can then be executed without authentication by injecting electromagnetic faults. The success probability of obtaining arbitrary code execution in this way increases as the size of the programmable flash grows, and on modern ECUs it is so high that an uninformed attack is successful within minutes.

No information about the software running on the target device is necessary for a successful attack. The map of the fault sensitivity can be obtained from another sample of the same MCU as the target one. Additionally, the attack was easily reproducible on multiple ECUs that were based on similar PowerPC MCUs with minimal changes to the exploit code and on an ARM-based evaluation board.

The equipment necessary to perform the presented attack is readily available, and the attack can be easily automated. Using a genetic evolution algorithm can reduce the time taken to find reproducible fault parameters from several days to less than one hour.

When applied to the real world, this attack can be used to reset stolen ECUs to a virgin state to resell them, pairing new keys to an immobilizer system, or aid in the development of further exploits by leaking firmware and restoring debug interfaces.

Responsible Disclosure

All affected OEMs were informed in April 2022 about this attack. Furthermore, we shared a draft version of this paper. We additionally disclosed this research to insurance companies and automotive software suppliers to discuss impact and mitigation strategies.

One target ECU from BMW is being phased out within 2022 and will then be no longer used in newly produced vehicles. The latest models of BMW vehicles make use HSMs and Secure Elements in their ECUs. Therefore, successful exploitation with the described attack might require additional steps.

References

1. AUTOSAR. *Specification of Secure Onboard Communication*, 2017. `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SecureOnboardCommunication.pdf` (accessed 2022-11-14).
2. Zero Day Initiative. *Pwn2Own Vancouver 2019: Wrapping Up and Rolling Out*, 2020 (accessed February 29, 2020). `https://www.zerodayinitiative.com/blog/2019/3/22/pwn2own-vancouver-2019-wrapping-up-and-rolling-out`.
3. Inc. Insurance Information Institute. *Facts + Statistics: Auto theft*, 2022. `https://www.iii.org/fact-statistic/facts-statistics-auto-theft` (accessed 2022-11-14).
4. ISO Central Secretary. Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services. Standard ISO 15765-2:2016, International Organization for Standardization, Geneva, CH, 2016.
5. ISO Central Secretary. Road vehicles – Unified diagnostic services (UDS) – Part 1: Application layer. Standard ISO 14229-1:2020, International Organization for Standardization, Geneva, CH, 2020.
6. James Gratchoff. Proving the wild jungle jump. Research project report, University of Amsterdam, 2015.
7. Tencent Keen Security Lab. *New Vehicle Security Research by KeenLab: Experimental Security Assessment of BMW Cars*, 2018. `https://keenlab.tencent.com/en/2018/05/22/New-CarHacking-Research-by-KeenLab-Experimental-Security-Assessment-of-BMW-Cars/` (accessed 2021-04-14).
8. Brian Ledbetter. *SA2 Seed Key*, 2022 (accessed January 30, 2022). `https://github.com/bri3d/sa2_seed_key`.
9. Brian Ledbetter. *VW Flashing Tools over ISO-TP / UDS*, 2022 (accessed January 30, 2022). `https://github.com/bri3d/VW_Flash`.
10. JinGen Lim. *UnlockECU: Free, open-source ECU seed-key unlocking tool.*, 2022 (accessed March 30, 2022). `https://github.com/jglim/UnlockECU`.
11. Dr. Charlie Miller and Chris Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. DEF CON 23 Hacking Conference. Las Vegas, NV: DEF CON, August 2015.
12. Pascal Nasahl and Niek Timmers. Attacking autosar using software and hardware attacks. In *escar USA*, July 2019. escar USA ; Conference date: 11-06-2019 Through 12-06-2019.
13. NXP. *Book E: Enhanced PowerPC Architecture*. NXP, 2002.
14. Colin O'Flynn. Bam bam!! on reliability of emfi for in-situ automotive ecu attacks. Cryptology ePrint Archive, Paper 2020/937, 2020. `https://eprint.iacr.org/2020/937`.
15. Pierre Lalet Philippe Biondi, Guillaume Valadon and Gabriel Potter. *Scapy*, 2018. `http://www.secdev.org/projects/scapy/` (accessed 2021-04-14).
16. NXP Semiconductors. *AN12092: Using the PASS module in MPC5748G to implement password-based protection for flash and debugger access*, 2018 (accessed January 30, 2022). `https://github.com/bri3d/sa2_seed_key`.
17. Yuefeng Du Sen Nie, Ling Liu. *FREE-FALL: HACKING TESLA FROM WIRELESS TO CAN BUS*, 2017. `https://www.blackhat.com/docs/us-17/thursday/us-17-Nie-Free-Fall-Hacking-Tesla-From-Wireless-To-CAN-Bus-wp.pdf` (accessed 2021-04-14).

18. Dieter Spaar. *Beemer, Open Thyself! – Security vulnerabilities in BMW's ConnectedDrive*, February 2015. `https://www.heise.de/ct/artikel/Beemer-Open-Thyself-Security-vulnerabilities-in-BMW-s-ConnectedDrive-2540957.html` (accessed 2021-04-14).
19. Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*, pages 25–35. IEEE Computer Society, 2016.
20. Jan Van den Herrewegen and Flavio D. Garcia. *Beneath the Bonnet: A Breakdown of Diagnostic Security*, volume 11098 of *Lecture Notes in Computer Science*, page 305–324. Springer International Publishing, 2018.
21. Jan Van den Herrewegen, David Oswald, Flavio D. Garcia, and Qais Temeiza. Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):56–81, Dec. 2020.
22. Nils Wiersma and Ramiro Pareja. Safety != security: On the resilience of asil-d certified microcontrollers against fault injection attacks. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 9–16, 2017.
23. Lennert Wouters, Jan Van den Herrewegen, Flavio D. Garcia, David Oswald, Benedikt Gierlichs, and Bart Preneel. Dismantling dst80-based immobiliser systems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):99–127, Mar. 2020.

Appendix

```
Machine Check Exception
Exception number:          1
Exception address:         0105D1EE
Stack pointer:             40006F98
R0     010F2FB8  R8      400070EC  R16  00000000  R24   400070EC
R1     40006F98  R9      013996A8  R17  00000000  R25   4004FAD8
R2     013DF918  R10     00000005  R18  00000000  R26   00000002
R3     02029200  R11     FFF1E400  R19  00000000  R27   00000002
R4     0000FFF1  R12     400070DC  R20  00000000  R28   0000E400
R5     00000000  R13     4001DD90  R21  00000000  R29   0000FFF1
R6     010F3130  R14     00000000  R22  00000000  R30   40007090
R7     0000FFF1  R15     00000000  R23  00000000  R31   4003EFA8
---------------------------------------------------------------
XER     00000000  CR   80000000  LR     010F2FB8
USPRG0 00000000  CTR  010F2EF4  IP     --------
---------------------------------------------------------------
SPRG0 00000000  SRR0   013D1FD6  IVPR 01000100  MSR   00000200
SPRG1 400200C8  SRR1   02029200  DEAR 00000000  PVR   81530000
SPRG2 00000000  CSSR0  00000000  ESR  00000000
SPRG3 00000000  CSSR1  00000000  MCSR 00088008
MCSSR0 0105D1EE  MCAR 00000078
MCSSR1 02021200
PID0  00000000
---------------------------------------------------------------
PIR 00000000

S T A C K T R A C E
> 0x010F2FB8
> 0x010F307A
> 0x010F1F1E
> 0x011281FC
> 0x0139957E
> 0x01023FFC
> 0x010243D2
> 0x0102523A
> 0x01025912
> 0x013981EC
> 0x011F0982
> 0x0103FA54
> 0x013D0C36
> 0x013D29E6
```

Listing 1.2. Example Stack-Trace

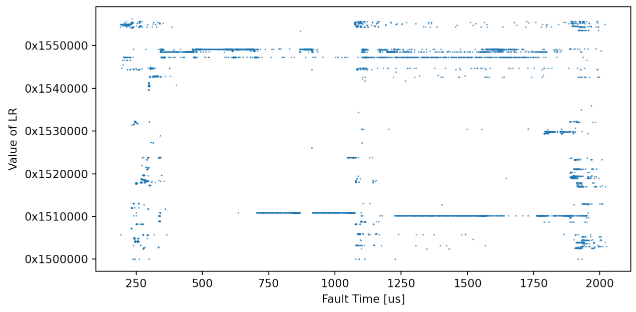

Fig. 5. Value of link register (LR) emitted on the stack traces caused by injecting a fault at different points in time. This gives an indication of where the ECU was running code from at any point in time between reception of the UDS request and emission of the UDS response.

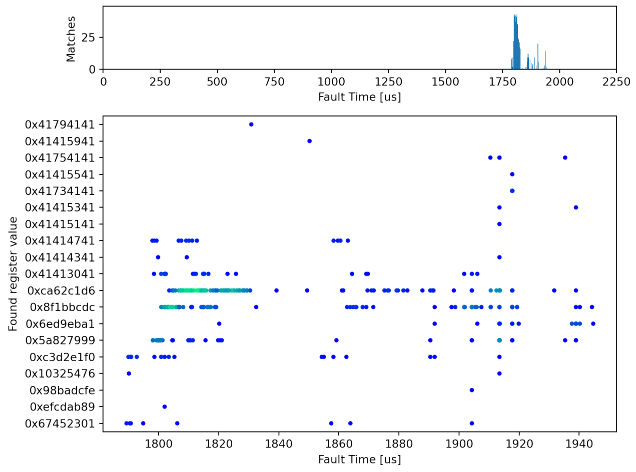

Fig. 6. Ten data words from the UDS request and nine constants from the SHA-1 algorithm were only found in the stack traces when the fault was injected 1.8ms after the UDS request was sent. The top bar plot shows the number of matches over the whole time axis, while the bottom plot shows a zoomed-in view of exactly which values were found in at least one of the registers in the stack trace. The first ten values (starting with hexadecimal 41) are the found words from the input pattern, while the last nine values are the constants from the SHA-1 algorithm.

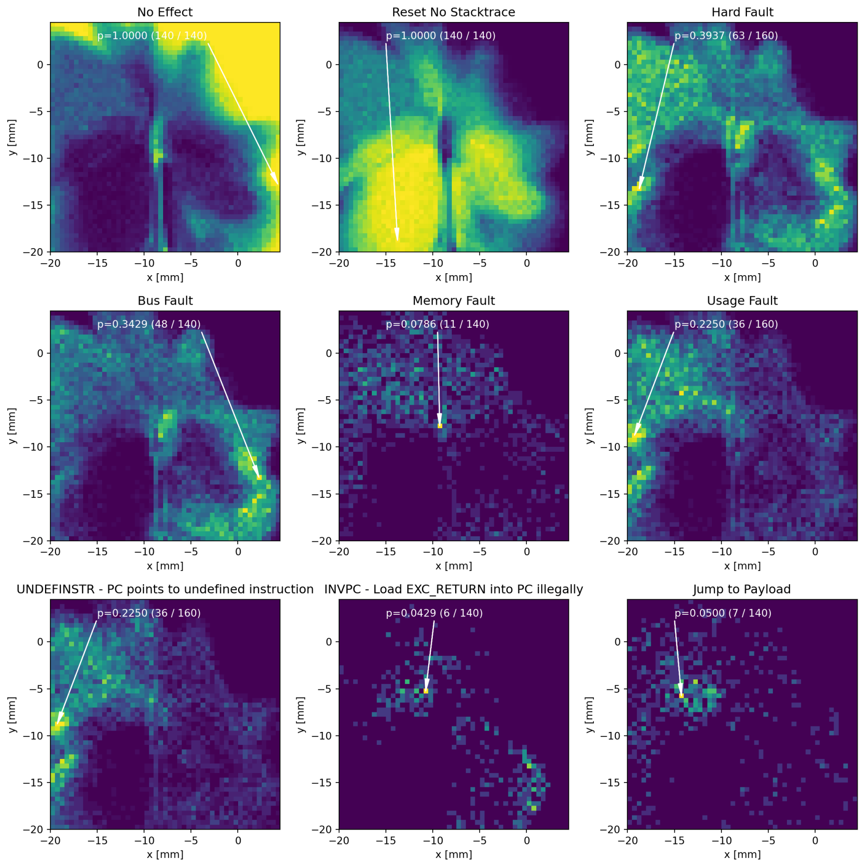

Fig. 7. Sensitivity of the different areas of the S32K148 MCU package to the fault with respect to different errors. These images were drawn from unbiased data with random and uniformly distributed fault parameters. p indicates the probability of a fault for the 0.5 $mm \times 0.5$ mm area pointed by the white arrows.