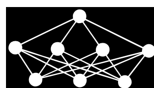

AN EVOLUTIONARY FAULT INJECTION SETTINGS SEARCH ALGORITHM FOR ATTACKS ON SAFE AND SECURE EMBEDDED SYSTEMS

E. Pozzobon,* N. Weiß*,* J. Mottok*,* V. Matoušek*†

Abstract: In this paper, we present a novel method for exploiting vulnerabilities in secure embedded bootloaders, which are the foundation of trust for modern vehicle software systems, by using a genetic algorithm to successfully identify the correct parameters to perform an electromagnetic fault injection attack. Specifically, we demonstrate the feasibility of code execution attacks by leveraging a combination of software and hardware weaknesses in the secure software update process of electronic control units (ECUs), which is standardized across the automotive industry. Our method utilizes an automated approach, eliminating the need for static code analysis, and does not require any hardware modifications to the targeted systems. Through our research, we successfully demonstrated our attack on three distinct ECUs from different manufacturers used in current vehicles. Our results prove that the use of a genetic algorithm for finding the fault parameters reduces the number of attempts necessary for a successful fault to obtain arbitrary code execution via "wild jungle jumps" by approximately 100 times compared to a naive random search.

Key words: *fault injection, security, genetic algorithm*

Received: April 28, 2023 **DOI:** 10.14311/NNW.2023.33.020
Revised and accepted: October 25, 2023

1. Introduction

Modern vehicles possess a unique threat landscape, distinct from that of other connected devices. Vehicles are 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 [4].

*Enrico Pozzobon – Corresponding author; Nils Weiß; Jürgen Mottok; University of Applied Sciences in Regensburg, Germany, E-mail: enrico.pozzobon,nils2.weiss,juergen.mottok@oth r.de

†Václav Matoušek; University of West Bohemia in Pilsen, Faculty of Applied Sciences, Czech Republic E-mail: matousek@kiv.zcu.cz

One popular form of physical attack against microcontrollers 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 [3].

1.1 Safe and secure microcontrollers

In many modern vehicles, security trust anchors are built with safe and secure microcontrollers, such as the MPC57xx series from NXP. Therefore, the manufacturer equipped them with a feature set rich of security functions and a dedicated embedded hardware security module (HSM) core. A wide range of security functionality allows high protection of the internal flash memories and debug interfaces of these processors.

1.2 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 [6, 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 original equipment manufacturers (OEMs) and suppliers in the world.

A simple challenge-response "Security Access" criptographic algorithm is used to enter the programming mode, which can be passed using (or reverse engineering)

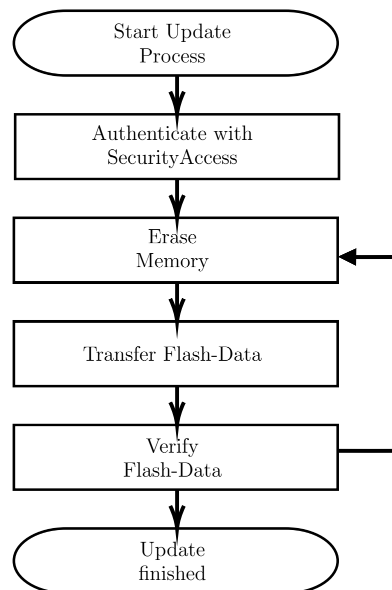

Fig. 1 *Typical flow chart for a secure software-update procedure of non-volatile memory, following the ISO 14229-1:2020 standard [6, p. 374].*

a repair shop tool. The main security measure against malicious code execution is the authentication of the firmware update binary via asymmetric cryptography.

A small portion of the flash memory is reserved for a read-only bootloader which is responsible for the update process, while the majority of the flash is used for the main application. When the main application is updated, the update service will erase the main application portion of the flash, flash the signed binary received over UDS, and verify its authenticity via asymmetric cryptographic authentication.

If the received binary is deemed authentic, it will be marked as executable and it will be executed on every successive boot. If the received binary was not authenticated, it will not be marked as executable and erased during the next attempted firmware update. Crucially, if the received update is not authentic, the electronic control unit (ECU) will remain in a state where it can not operate normally until a signed firmware updates is received.

1.3 Structure of this paper

This paper is structured as follows. Section 2 states related work in the area of embedded and automotive security, focusing on fault injection attacks. Section 3 summarizes our contributions. The used test setup and our target ECUs are introduced in Section 4. Section 5 explains information-gathering possibilities through fault injection attacks on black box targets. A novel fault search algorithm is presented in Section 6. Section 7 describes vulnerabilities introduced by fault injection attacks and demonstrates the exploitation of secure automotive bootloaders in real-world targets. Section 8 discusses the application of the presented attack and the fault search algorithm to different ECUs and instruction set architectures (ISAs). Section 9 concludes this paper.

2. Related work

In the paper "BAM BAM!! On reliability of electromagnetic fault injection (EMFI) for in-situ automotive ECU attacks [11]", 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 [15] 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. Instead of attacking the DCF system, this paper targets automotive bootloaders to broaden the number of affected targets.

Wouters et al. [16] 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. Our work uses a different fault injection methodology and targets german car manufacturers instead.

Attacks against internal bootloaders of three different MCUs were demonstrated and summarized by Van den Herrewegen et al. [14]. 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 [10]. By leveraging fault injection weaknesses in the ARM ISA they could corrupt a `memcpy` operation to perform a jump into writable RAM memory. We want so show that EMFI is effective against other architectures too.

Maldini et al. [9] applied genetic algorithm to EMFI on a pinata target board, with the intention of breaking the WolfSSL implementation the SHA-3 algorithm, successfully showing the advantage of using genetic algorithms for this purpose. We aim to present similar results in a real world target rather than a developer board.

Carpi et al. [2] applied genetic algorithm to VCC fault injection, which presents a smaller search space than EMFI because the it is not affected by the spatial position of the injection tool.

3. Contribution

In this paper,

- we present *a novel algorithm (called EFISSA)* using a feedback-channel to efficiently estimate which glitch parameters cause so-called wild jungle jumps,
- we show the *performance of this algorithm* on PowerPC (PPC) – and ARM-based automotive processors for safety-critical applications,
- we demonstrate the application of this algorithm through the *exploitation of three different* secure bootloaders in real-world *ECUs*, following the ISO 14229-1:2020 standard [6],

The presented attack gains *code execution* through fault injection *without* the requirement of *binary analysis or reverse engineering* and can be applied fully automated to black box real-world targets.

This paper documents successful exploitation of program-counter glitches on PowerPC architectures, for the first time. Our novel algorithm makes wild jungle jumps applicable on real-world targets. Until now, wild jungle jumps were only exploitable in laboratory environments and considered impossible in practice [13].

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, 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. Another advantage of having an automated test setup is to parallelize the attack to multiple ECUs at the same time by building multiple setups.

To reduce the manual work necessary, we only performed our tests with two coils included in the ChipSHOUTER kit: a 1 mm diameter core clockwise wound coil, and a 1 mm 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 and composed of the following items:

- *USB-to-CAN* – for controller area network (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
- *ICEBreaker field-programmable gate array (FPGA)* board – for consistently triggering the glitch on a specific CAN message, and manipulating the timing of the electromagnetic fault
- *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 ±0.01 mm. The diagnostic CAN interface of the ECU is connected via

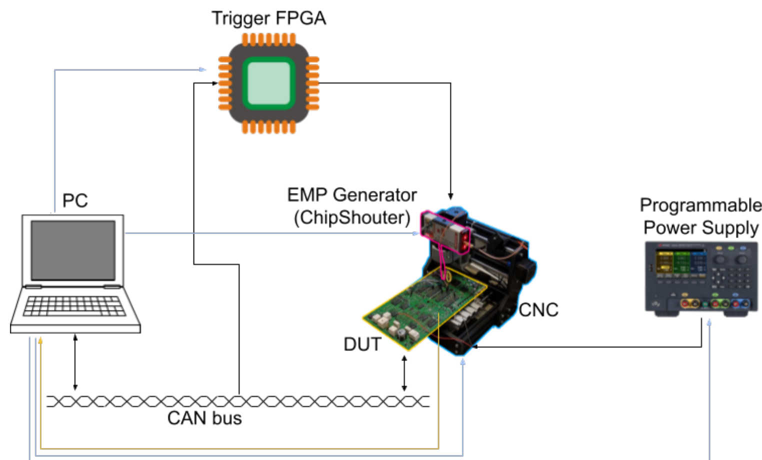

Fig. 2 *Diagram of our automated test setup.*

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 trigger signal with a configurable delay and duration. 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 specific components of the test setup were chosen because they were either readily available in a laboratory environment or cost effective to purchase. To build a cheaper setup that can easily scale, the ChipSHOUTER can be replaced with a ChipSHOUTER-PicoEMP [12] and the programmable power supply can be replaced with a simple 12V wall adapter and a relay.

The software used to control the setup was written in the Python programming language, using Scapy for the CAN and UDS communication [1]. 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 an ECU that 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. 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. Later on, the attack was tested successfully on different ECUs from other manufacturers, some of which did not have UART logging.

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 ISO 15765-2 transport protocol (ISOTP) consecutive frame of a `TransferData` UDS request but before the corresponding response, as seen in Fig. 3 [5]. This specific trigger attempts to inject the glitch during the processing of the `TransferData` UDS request.

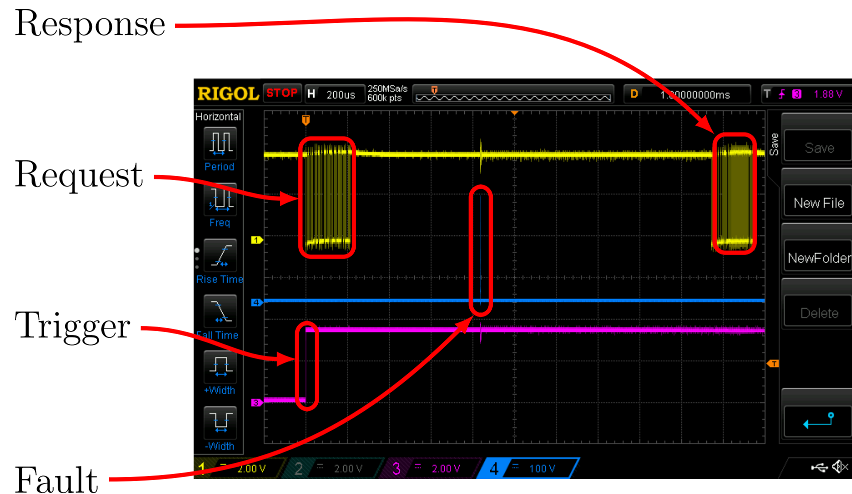

Fig. 3 *A snapshot of the oscilloscope screen during a fault injection attack. The yellow line represents the CAN protocol, and shows the request and response ISOTP messages. The magenta line is the trigger from the FPGA, which detected the searched CAN frame. The blue line shows the voltage spike sent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

5. Information gathering

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

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.

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. The special registers 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.

As illustrated in 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.

6. Fault search algorithm

Under specific conditions, the injection of a fault in the target core can result in a disruption of the execution flow and an unintentional branch to unsigned code. This phenomenon typically arises from an undetected memory read error during instruction fetch, which subsequently corrupts to an instruction that alters the program counter, either directly or indirectly through manipulation of the link register or stack pointer.

In general, it is possible to randomly inject faults until one just happens to affect the program counter in just the desired way. However, the search space for the parameters of injected faults is quite large and a random search algorithm for the right parameters can take from a few hours to months depending on the target processor and the rate at which faults can be generated.

6.1 Definition of the search space

The search space of all the possible faults that can be injected with our setup corresponds to the multi-dimensional space (x, y, z, c, i, t, d) 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
- d: time *offset* from trigger

In addition to the above-mentioned parameters, the input data sent to the target device before the injection also affect the state at the moment of the attack, so the memory state m of the target should also be considered as part of the search space for a fault.

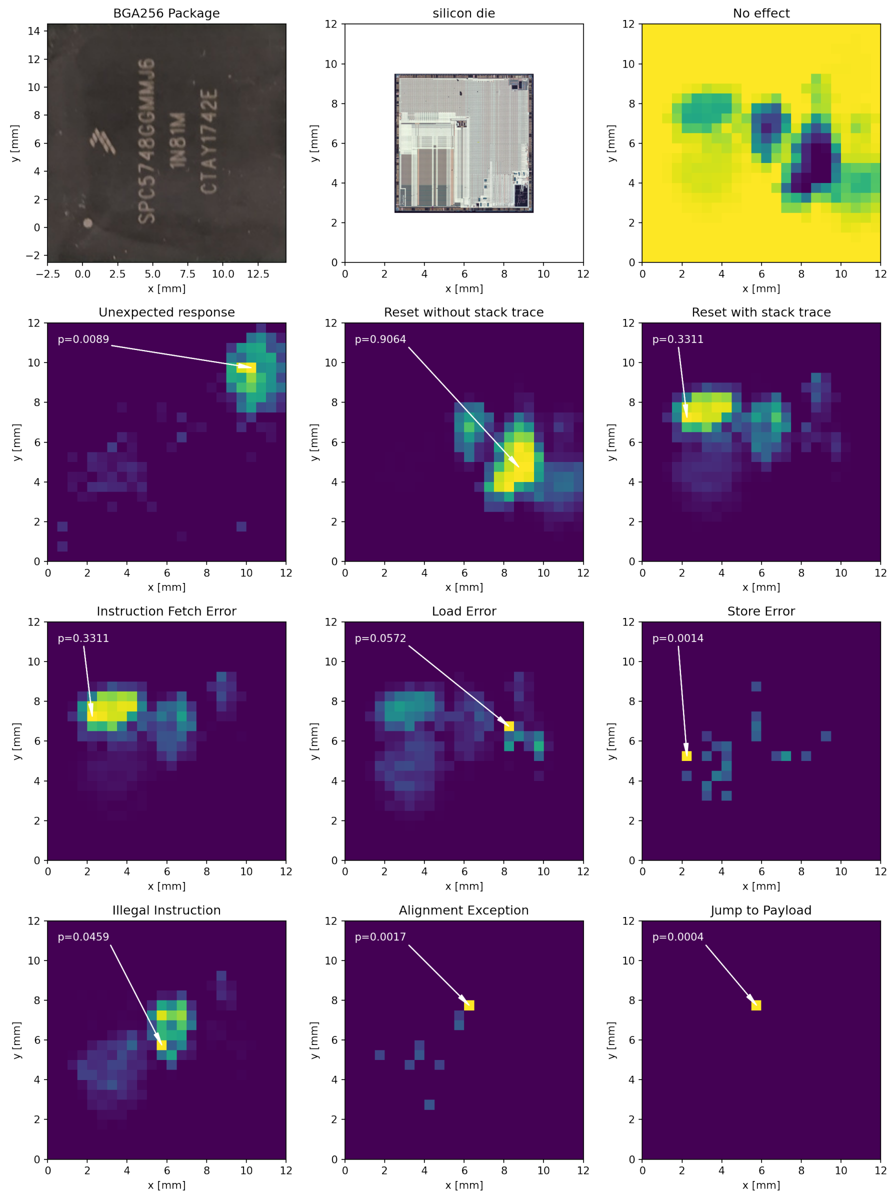

Fig. 4 *Sensitivity of the different areas of the MPC5748G MCU package to the fault with respect to different errors. These images were drawn from the dataset* $\mathbf{X}$ *unbiased data with random and uniformly distributed fault parameters, using the 1 mm core diameter counter-clockwise wound coil. 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.*

A subset of the parameters (x, y, z, c, i, t) are only tied to the hardware of the target processor and do not depend on what software the target is running. Only the time of injection (o) after the trigger signal and the memory state (m) of the target is dependent on the particular application the target is running.

Our objective is to build an algorithm that finds the correct fault parameters by continuously generating new parameters, testing them on the target, and using the stack traces received from the target to refine them.

It is not guaranteed that stack traces will be enabled or present on the target, but it is usually possible to purchase an identical microcontroller to the one used on the target and flash it with an example program with stack traces enabled, thus creating a "test dummy". By using the same search algorithm on the test dummy, it is possible to find at least the correct parameters that are tied to the hardware, thus leaving a much smaller search space when it comes to performing the attack on the real target. Ideally, such a test dummy would be created by using the same exact MCU package and printed circuit board (PCB) as the real target to reduce the possible differences between the real target and the test dummy to a minimum: a valid option here is purchasing a malfunctioning ECU and replacing the MCU with a new blank one to use as a test dummy.

6.2 Overview of the algorithm

Given the input $\overline{\sigma} = (x, y, z, c, i, t, d, m)$ being the parameters of the fault, our glitch setup will return some output feedback b received from the target, containing for example the UART log and/or the UDS response. This output feedback is to be parsed by a reward function $f(b)$ which will assign it a numerical rating r depending on how desirable the result was. For example, causing the target to reset with a stack trace is more desirable than a reset without a stack trace, so the former case will be assigned a higher rating than the latter. This is reasonable because a reset without a stack trace can be generated when the target was hit "too hard" and/or the power supply circuitry was hit, which is of no use for an attacker.

The general workflow of the system is the following:

1. The search algorithm generates a tuple $\overline{\sigma}$ of fault parameters (x, y, z, c, i, t, d, m)
2. The target is brought into state m.
3. The glitch setup injects the fault and returns some log.
4. The reward function parses the log and returns a rating r to the search algorithm.
5. The search algorithm updates its internal state in such a way as to increase the likelihood that the next generated fault will have a high rating.

6.3 EFISSA

The search algorithm we developed for this task, named EFISSA (Evolutionary Fault Injection Settings Search Algorithm), is a genetic algorithm and as such is inspired by the process of natural selection.

A pool of fault parameters tuples, called the "population", is initialized with random values at the start of the algorithm. Each tuple of fault parameters is tested by the glitch setup, and the rating r returned by the reward function is added to the fitness score of that tuple. The fitness score of each tuple is used to decide how many "offspring" that tuple will have in the next generation. These offspring are simple copies of the tuple which have been mutated by altering some bits of its binary representation (which represents their genome) with a small mutation probability applied to each bit. The tuples with low fitness scores are removed from the current population, and new randomized tuples are introduced in the population if the population size becomes too small.

Optimizing the path taken by the glitch setup to navigate through the population of fault parameters is crucial to increase the fault rate and shorten the time necessary to find a successful fault. In particular, moving the fault injection coil on the XYZ table takes an amount of time proportional to the distance between the current position and the desired position; changing the voltage on the ChipSHOUTER takes a fixed amount of time of around 1 second; while changing the parameters on the FPGA only takes a couple of milliseconds. Because of this, a distance function between two tuples is defined as the expected time taken to change the settings between the two, and the array of faults in the population is ordered to minimize the sum of the distances between consecutive faults before being sent to the glitch setup.

6.4 Definition of the reward function

The reward function parses the feedback from the target after a fault to generate a rating r for the fault. It is worthwhile to remember that the same fault parameters tuple can produce different results due to noise and manufacturing tolerances in the glitch setup, so it doesn't always map to the same rating.

The reward function needs to be defined for every target architecture and should return a rating that is proportional to the correlation of the obtained feedback to the desired result of the fault.

Taking Fig. 4 as an example, since there is a high correlation between the parameters that caused an "alignment exception" to the ones that caused a "jump to payload" (the desired result), then "alignment exception" should have a high rating.

By the same principle and same figure, we can derive the following sequence of ratings that should be returned by the reward function $f(feedback)$: $f(\text{no effect}) < f(\text{reset without stack trace}) < f(\text{reset with stack trace}) < f(\text{illegal instruction}) < f(\text{alignment exception}) < f(\text{jump to payload}) = +\infty$. $f(\text{jump to payload})$ is set to $+\infty$ because it is the desired result and represents a successful fault, therefore it receives the highest possible reward.

Note that we define a "jump to payload" event to happen when the Program Counter jumps to any location in the erased application flash which can be manipulated by an attacker using repair shop tester tools as explained in Section 7.1.

In general, without having to collect the data necessary to plot the figure, we can assign ratings according to these categories:

1. *No fault* (lowest rating): faults that produce no noticeable effect whatsoever should be assigned the lowest rating because in all likelihood they are not positioned correctly to affect the execution of the target.

2. *Reset* (low rating): faults that cause an instantaneous reset of the target without any output or stack trace should receive a low rating, because they are probably hitting the target too hard or hitting the power supply circuitry of the die, rendering it incapable of producing a stack trace.

3. *Generic exception* (medium rating): a fault that caused a generic stack trace (not Illegal Instruction) should be assigned a middle score, since it was strong enough to affect the target but not strong enough to instantly reset it, and it hit the die close enough to the CPU to have visible effects on the execution.

4. *Illegal instruction* (high rating): a stack trace containing an "Illegal Instruction" exception should always receive a high rating because the majority of faults that corrupt an instruction fetched from memory will result in an invalid instruction in most architecture.

5. *Jump to payload* (highest rating): a fault that causes provable execution of unsigned/unreachable code should receive the highest rating, to ensure that the genetic algorithm will continue to produce mutated faults that potentially achieve a higher probability of a successful fault.

6.5 Tuning of the evolutionary algorithm parameters

The choice of the parameters is a critical part of any machine learning algorithm. For an evolutionary algorithm like EFISSA, the most important parameters to tune are population size, mutation rate, selection mechanism and reward function.

Testing a new set of parameters while tuning them is slow on the real hardware setup, since the time necessary to inject each fault is quite long (about half a second to a couple of seconds), so each test can easily take hours. To increase the speed at which these parameters can be tested, a simulated environment was built out of the data collected during the injection of millions of randomized faults. The simulated EFISSA environment can inject millions of faults per second on a virtual target while still calculating how much time it would have taken to execute the same faults on real hardware, by estimating the time it took to move the injection head to the new position and changing the configuration of the ChipSHOUTER.

This dataset, named $\mathbf{X}$, is the result of running 10256642 faults over several months with uniformly distributed fault parameters on a real automotive hardware target. It contains the fault parameter tuples $\overline{\sigma}$ as well as the corresponding output of the communication interfaces of the target to allow to identify if a fault happened and which kind of exception fault handler was executed.

When a fault is injected in the simulated environment with a given fault parameters tuple $\overline{\sigma}_{\mathrm{T}}$, the virtual target returns the result of a fault injected with a fault parameter tuple $\overline{\sigma}_{\mathrm{S}}$, sampled randomly from the $\mathbf{X}$ after weighting the probability of each point according to a multivariate Gaussian distribution centered on $\overline{\sigma}_{\mathrm{T}}$. Since $\mathbf{X}$ is collected from real hardware on random, uniformly distributed fault parameters, $\overline{\sigma}_{\mathrm{S}}$ is typically a point in the vicinity of $\overline{\sigma}_{\mathrm{T}}$, thus ensuring that the

value returned by the simulated environment correlates to the value that would be returned by the real experiment.

It is important to remember that when injecting a fault with the same fault parameters multiple times, the result is usually not the same. Due to this, it is not sufficient to return the result for the closer tuple of parameters to $\overline{\sigma}_{\mathrm{T}}$ in $\mathbf{X}$, since this would cause the success probability of some tuples to be 100% in the simulated ECU, which is not realistic.

After experimentation on the simulated environment, the algorithm was tuned as follows:

- The population size was set to 100, though every tuple is tested 10 times every generation to increase the number of faults per second since changing the fault parameters costs a lot of time. A larger population size leads to generations taking too long to be fully tested, slowing down the algorithm.

- The bit mutation probability was set to 0.01, meaning each bit has a 1% probability of being flipped when an offspring is created by copying a parent.

- No cross-over was used because the implementation of a cross-over functions did not lead to a noticeable improvement in performance.

- An aging coefficient of 0.9 was implemented, meaning that the fitness value of each tuple in the population was multiplied by 0.9 at every generation, ensuring that newly generated solutions will have a fair chance at competing with tuples that were in the population for a long time.

- Elitism was implemented, preserving the tuples of fault parameters which resulted in the 10 highest scores according to the reward function.

- The reward function still needs to be tuned for each different target, but in the simulations it was seen that the best results are obtained by granting the categories defined in Section 6.4 the ratings:

  - $f(\textit{no fault}) = 0$
  - $f(\textit{reset}) = 1$
  - $f(\textit{generic exception}) = 10$
  - $f(\textit{illegal instruction}) = 100$
  - $f(\textit{jump to payload}) = \infty$

  These rewards are added to the fitness of the tuple in the population.

- The selection function (which chooses which tuples are selected for having offsprings in the next generation, and how many offsprings they will produce) was chosen to be random with a selection probability proportional to the logarithm of the fitness.

6.6 Performance

For the purpose of discussion of the performance of the presented algorithm, any jump over the application flash of the processor caused by a fault will be considered a success. This is because the content of the application flash can be controlled by an attacker as explained in Section 7.1.

We observed that the success probability of a fault attack is much higher on development boards rather than automotive targets, possibly due to the large ground planes and thicker copper used in automotive ECUs. The presented results were obtained by attacking the bootloader in a modern ECU which emitted stack traces upon encountering an exception interrupt. On said ECU, the MCU that the fault injection was performed on was a MPC5748G.

Fig. 5 shows the probability of getting a success given that the algorithm has been running for a given number generated faults.

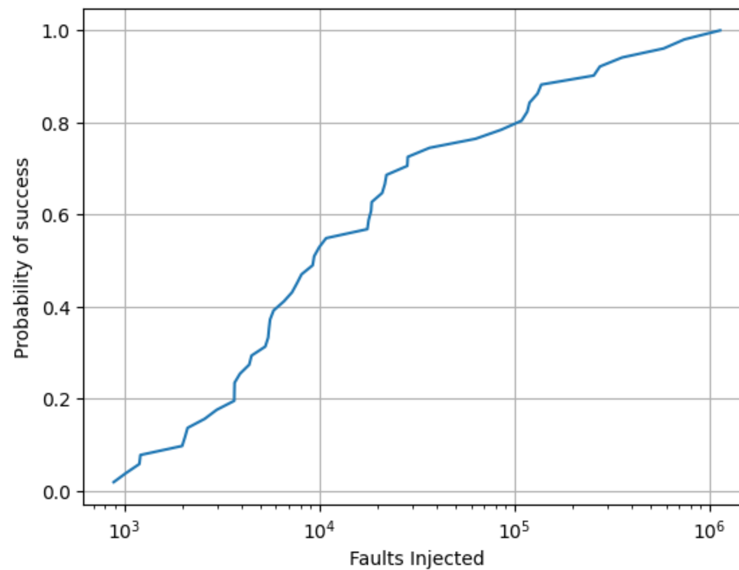

Fig. 5 *Cumulative distribution function of the probability of finding a fault that causes a jump of the program counter to any address within the application flash on an automotive bootloader as a function of the number of faults generated by the EFISSA algorithm. In total 4486208 faults were injected on the target to generate this figure.*

Once a success is found by the algorithm, the successes can be replicated with an average probability of 0.674%. In other words, on average one every 148 faults injected with the parameters found by EFISSA resulted in a success, or about one success per minute with an average fault injection rate of 2 faults per second. On average, these successful parameters are found after injecting less than 10^4 faults. Conversely, when using a random search algorithm, the probability of obtaining a successful fault was around once every 10^6 attempts.

7. 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.

If the contents of the large application flash memory can be controlled by an attacker, the random jumps inside the flash obtained through fault injection represent a vulnerability because there seems to be nothing preventing the MCU from executing unsigned code.

7.1 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, 7]. 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 unsigned firmware can not normally be executed, since signature checks will fail, but the code can still be reached by jumping to it using the fault injection attack described earlier.

7.2 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  // Jump to _start
.endr
_start:
        // The actual exploit code is written here
        // Can be a simple ''Hello, World!'' for demonstration
        // Or malicious code to help car theft
```

Listing 1 *Example GNU assembler code which generates a long PPC nopslide which interleaves one branch instruction every 113 NOP instructions.*

Since PPC variable length encoding (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.

By applying faults during the execution of some UDS service, which can be requested with a CAN frame which will be the trigger of the fault, it is possible to 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 the exploit code.

To demonstrate that arbitrary code execution was achieved, a simple "Hello, World!" example was written in assembly and placed in the payload. A real world attacker would use malicious code as payload. For example, a routine to dump the firmware or one to remove the signature check of the firmware from the bootloader.

After using EFISSA, presented in Section 6.3, we obtained fault parameters that granted a decent success probability (0.674%) of executing our unsigned payload. Without the search algorithm, our fault led to code execution around once every 10^6 attempts.

8. 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 selection with EFISSA 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 EFISSA on an ARM S32K148 evaluation board. It was found that the ARM processor architecture is significantly more susceptible to wild jungle jumps into writable memory areas, compared to PowerPC processors.

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).

9. Conclusions

We demonstrated the efficient application of fault injection attacks to obtain code execution through program counter manipulation on different real-world targets. Our novel algorithm for fault parameter search makes this kind of fault injection attack feasible on black-box 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 making use of a genetic algorithm to search for fault parameters 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 cheap and readily available, and the attack can be easily automated. Using the presented algorithm (EFISSA) reduces the time taken to find a reproducible fault 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.

References

[1] BIONDI P., VALADON G., LALET P., POTTER G. *Scapy*. 2018. http://www.secdev.org/projects/scapy/ (accessed 2021-04-14).

[2] CARPI R.B., PICEK S., BATINA L., MENARINI F., JAKOBOVIC D., GOLUB M. Glitch It If You Can: Parameter Search Strategies for Successful Fault Injection. In: *Smart Card Research and Advanced Applications: 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, Berlin, Germany: Springer-Verlag, 2014, pp. 236–252. Available also from: https://doi.org/10.1007/978-3-319-08302-5_16 doi: 10.1007/978-3-319-08302-5_16. ISBN 978-3-319-08301-8.

[3] GRATCHOFF J. *Proving the wild jungle jump*. NL: University of Amsterdam, 2015. Research Project Report. Available also from: https://rp.os3.nl/2014-2015/p48/report.pdf.

[4] INSURANCE INFORMATION INSTITUTE I. *Facts + Statistics: Auto theft*. 2022. https://www.iii.org/fact-statistic/facts-statistics-auto-theft (accessed 2022-11-14).

[5] ISO CENTRAL SECRETARY. *Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services*. Geneva, CH: International Organization for Standardization, 2016. Standard ISO 15765-2:2016. Available also from: \url{https://www.iso.org/standard/66574.html}.

[6] ISO CENTRAL SECRETARY. *Road vehicles – Unified diagnostic services (UDS) – Part 1: Application layer*. Geneva, CH: International Organization for Standardization, 2020. Standard ISO 14229-1:2020. Available also from: \url{https://www.iso.org/standard/72439.html}.

[7] LEDBETTER B. *SA2 Seed Key*. 2022 (accessed January 30, 2022). https://github.com/bri3d/sa2_seed_key.

[8] LIM J. *UnlockECU: Free, open-source ECU seed-key unlocking tool.* 2022 (accessed March 30, 2022). https://github.com/jglim/UnlockECU.

[9] MALDINI A., SAMWEL N., PICEK S., BATINA L. Genetic Algorithm-Based Electromagnetic Fault Injection. In: *Genetic Algorithm-Based Electromagnetic Fault Injection*, 2018, pp. 35–42. doi: 10.1109/FDTC.2018.00014.

[10] NASAHL P., TIMMERS N. Attacking AUTOSAR using Software and Hardware Attacks. In: *escar USA*, 2019. escar USA; Conference date: 11-06-2019 Through 12-06-2019.

[11] O'FLYNN C. *BAM BAM!! On Reliability of EMFI for in-situ Automotive ECU Attacks*. 2020. https://eprint.iacr.org/2020/937. Cryptology ePrint Archive, Paper 2020/937https://eprint.iacr.org/2020/937.

[12] O'FLYNN C. *ChipSHOUTER-PicoEMP*. 2021. https://github.com/newaetech/chipshouter-picoemp (accessed 2022-11-14).

[13] SPENSKY C., MACHIRY A., BUROW N., OKHRAVI H., HOUSLEY R., GU Z., JAMJOOM H., KRUEGEL C., VIGNA G. Glitching Demystified: Analyzing Control-flow-based Glitching Attacks and Defenses. In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 400–412. doi: 10.1109/DSN48987.2021.00051.

[14] Van den HERREWEGEN J., OSWALD D., GARCIA F.D., TEMEIZA Q. Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*. 2020, 2021(1), pp. 56–81, doi: 10.46586/tches.v2021.i1.56-81.

[15] WIERSMA N., PAREJA R. 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)*, 2017, pp. 9–16. doi: 10.1109/FDTC.2017.15.

[16] WOUTERS L., Van den HERREWEGEN J., GARCIA F.D., OSWALD D., GIERLICHS B., PRENEEL B. Dismantling DST80-based Immobiliser Systems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*. 2020, 2020(2), pp. 99–127, doi: 10.13154/tches.v2020.i2.99-127.