Safety != Security

On the resilience of ASIL-D certified microcontrollers against fault injection attacks

***Abstract*—With the ever increasing amount of electronic components in vehicles and in particular the amount of complex autonomous actions that these components perform, security topics become more and more important in the automotive industry. Not only from a business point of view, as valuable IP assets are contained within these components, but also from a safety point of view, especially when vulnerabilities lead to remotely exploitable attacks. Logical attacks in this field are abundant, but attacks using hardware centered techniques such as fault injection are underrepresented. Researchers and professionals often pay no attention in these attacks because they require physical access to the chip, ignoring that the assets obtained (e.g. firmware, keys, etc.) can be used later to prepare a remote attack.**

This work aims to address the lack of attention on fault injection attacks by investigating two modern microcontroller units that receive the highest safety assurance rating (ASIL-D) of the ISO 26262 automotive standard. This is done in both a theoretical characterization setup and a more realistic setup where debugging interfaces are targeted. The results obtained from these setups show that the mechanisms implemented to adhere to this maximum safety rating do not adequately protect against fault injection attacks and are therefore insufficient to ensure security by themselves – additional countermeasures are required. Each setup required approximately one week of preparation, but once the attacker finds the optimal fault injection parameters, the attack can be repeated in less than an hour. We provide some recommendations on what type of countermeasures should be considered to improve the security with respect to fault injection attacks and also provide several pointers to continue the security research in this area.

***Keywords*-fault injection; power analysis; ASIL; ISO-26262; safety mechanisms; countermeasures; lockstep; JTAG**

I. Introduction

Microcontroller units (MCU) form the foundation of the modern car. They are used to implement functionalities that can be as simple as opening the window when a button is pressed, or as complicated as Vehicle to Vehicle communications and autonomous driving systems. They can perform non-critical tasks such as controlling the windscreen wipers or extremely critical ones like making sure airbags are deployed in case of an emergency.

Over the last two decades the number of electronic controllers in vehicles has increased, making evident the need of ensuring the reliability and safety of these MCUs. The ISO 26262 standard [1] – in particular part 10, Annex. A and part 5, Annex. D – recognizes the pivotal part that MCUs play in the ecosystem of a vehicle and establishes a framework for the development of safe automotive electronic systems. Part 5 of the standard gives a number of examples of safety mechanisms that can be implemented to mitigate different electric faults that could affect the operation of an electronic system and consequently threaten the safety of the vehicle's passengers. Although these examples and recommendations intend to protect against faults occurring during normal use of the vehicle, some of these safety measures are also often recommended in the security literature as countermeasures against fault injection (FI) attacks (e.g. [2]). Previous work [3] shows that voltage glitching can be prevented by these kinds of mechanisms. It is therefore easily taken for granted that MCUs implementing ISO 26262 recommendations (i.e. ASIL-D chips) are also protected against FI attacks.

This paper summarizes our investigations into the effectiveness of the ISO 26262's recommendations as a means to resist against FI attacks. We first determine the resistance to FI of two ASIL-D certified MCUs and a standard (non-ASIL) automotive MCU in a controlled test environment. Next, we explore the risks of FI on ASIL-D chips by attacking the debug interface protection mechanism in our targets. Finally, we discuss the consequences and threats these attacks pose in the automotive context and provide recommendations for mitigating the aforementioned threats.

II. Automotive context, ISO 26262 and fault injection

Automotive standards and literature, especially with regards to hardware, primarily deal with safety. In this context, faults are considered as an unintended, abnormal and often random condition caused by the hostile environment in which the hardware resides (e.g. high temperatures, noisy power supply, vibrations, electromagnetic (EM) pulses produced by coils, etc.). ISO 26262 provides recommendations to mitigate and minimize the impact of these unintentional faults and defines the Automotive Safety Integrity Level (ASIL) risk classification scheme from level A through D (see Table I). ASIL-D complying MCUs are required to adhere to the most stringent requirements on safety and fault-tolerance in order to achieve absence of unreasonable risk. A fifth level, QM, is defined for parts that only require basic quality management. FI is mentioned in the ISO 26262 only as a method of testing the compliance of the ASIL requirements.

When shifting to the domain of security, faults are no longer the effect of random and unintended events. They are

Table I
ASIL CATEGORIES

Category	QM	ASIL-A	ASIL-B	ASIL-C	ASIL-D
Assurance level	Low	⟶			High

the result of *glitches* in the environment that are deliberately placed, precisely timed and carefully tuned by an attacker with the purpose causing unintended behavior – the *fault* – which is leveraged to compromise the chip security. These FI attacks are in general used to alter the program flow or the processed data. If this is possible, an attacker could, for example, bypass the Secure Boot process by skipping the signature verification and executing unauthenticated code [4], bypass the authentication of the UDS protocol to extract the firmware or recover cryptographic keys using a Differential Fault Analysis (DFA) attack [5], [6], [7].

For this work, two techniques of injecting faults have been considered: voltage glitching – a fast variation of the power supply voltage – and transient EM glitching – a strong and localized EM pulse. A voltage glitching device is cheap (tools can be bought for even less than 200) but the attack is not localized, as it affects the entire chip, and requires modifying the target board by removing the capacitors. EM glitching on the other hand requires no modifications on the board and the glitches are local (i.e. they affect a small area of the chip) but the equipment required is more expensive.

III. TARGETS

To investigate the effectiveness of safety mechanisms as countermeasures against FI attacks, two ASIL-D certified MCUs that implement these mechanisms have been selected, from two different manufacturers. Although we believe that we would have obtained similar results with any other ASIL-D MCU in the market, we omit the chip models because we are still in the process of disclosing the findings to the manufacturers in a coordinated manner.

The first target, codenamed ASILD1, implements an ARM Cortex-R4 core and the second target, codenamed ASILD2, a PowerPC e200. These two targets were selected because they represent two of the most prevalent architectures in the safety electronic industry. For reference and comparison, a non-ASIL target has been chosen, code-named QM1. This target implements an ARM Cortex-M0 and can be used for simple applications which require only the basic QM rating.

Of all the safety mechanisms implemented in ASIL-D MCUs, we are only interested in investigating the ones that have an effect on transient faults as they could also mitigate the glitches used by an FI attacker. In both selected targets these mechanisms include a dual core CPU in lockstep configuration (or 'Simple Time Redundancy with Comparison') and memories with error correction codes (ECC) and parity bits, as recommended by ISO 26262 part 5.

A. FI tools

Commercially available hardware and software is used for injecting glitches. The central piece to the hardware setup is the VCGlitcher, an FPGA powered device capable of generating two types of glitches: voltage glitches with 2ns of precision; or input to an EM pulse emitter device for the purpose of EM glitching. During the characterization experiments where the target is fully under our control and running our firmware, a GPIO pin of the target is used to generate a signal that triggers the VCGlitcher. In the experiments where the target is not controlled, the icWaves tool is used. This device analyzes the power consumption of the target and generates a trigger signal when a programmable reference pattern is detected. The setup is controlled by a Windows host machine running the Inspector FI software.

B. Target preparations

Standard low-cost development boards are used as target for the three MCUs under test. For voltage glitching, the power traces of the printed circuit board (PCB) are modified where needed. The purpose of this modification is to isolate the MCU power supply from the rest of the circuit, so the voltage glitch affects only the MCU and not the other components on the PCB. Additionally, the decoupling capacitors of the MCU are removed as they affect the shape of the voltage glitch. For EM glitching, no modifications to the boards are needed.

In order to determine the effect of the glitch, in particular whether or not it caused a useful fault, targets are programmed with software that outputs the contents of several internal registers over UART. This allows us to determine whether or not a particular safety mechanism has been triggered.

C. Glitch parameters

The effectiveness of a glitch depends on the moment in time when the glitch is injected and the shape of the glitch. These can be modeled through parameters like the glitch length, glitch intensity (voltage or EM field) and time offset from the trigger signal. In the case of EM glitching, the point on the chip surface where the EM field is applied is also important.

A glitch can have various effects on a target, depending on the aforementioned parameters. A glitch is considered successful when it causes a fault in the target that results in the behavior that the attacker is looking for (e.g. bypassing a check). A glitch can also result in crashes or unexpected behavior. In this work, special attention is paid to glitches that are detected (or not) by the safety mechanisms under investigation. When a glitch is mentioned as detected, it means that it was detected by one or more of these mechanisms. Classification is based on internal error registers and externally observable error output pins.

The methodology used for finding the optimal parameters – that is, those with the highest success rate – involves running multiple FI campaigns. In each campaign, thousands of glitches are attempted while varying the parameters within a certain range. The initial FI campaign runs on very broad parameters. The parameters of the successful glitches found are used to restrict the search range in the next campaign and after few iterations of this process the optimal parameters are found.

IV. Characterization

The general effectiveness of the safety mechanisms in preventing FI is investigated in a characterization test. This test involves two experiments in a controlled environment: a very simple setup featuring a set of unrolled addition instructions and a more intricate scenario that employs a conditional branch.

A. Scenario 1 – `unroll`

In the first experiment, `unroll`, a sequence of increment instructions is performed on the same CPU register. This should produce a predictable final value in the counter register. The final value of the register is sent over a serial connection at the end of the sequence together with the internal state of the MCU and its fault detectors. A pseudo code version of this setup is given in Listing 1.

```
trigger_up()
asm(add r1 #1)
asm(add r1 #1)
...
asm(add r1 #1)
asm(add r1 #1)
trigger_down()
send_serial(r1)
```

Listing 1. `unroll` scenario pseudo code

A successful glitch affects the value of `r1` by altering the content of the register itself or by changing the code flow and skipping one (or more) increment instruction(s), without triggering the fault detectors.

B. Scenario 2 – `auth`

`unroll` is one of the most basic experiments that can be performed and a good starting point for characterizing MCUs. The goal of the test is to confirm that the target is sensitive to FI. If successful glitches are found, additional experimentation should be done involving more realistic scenarios. The additional experiment chosen in this work is a conditional branch setup – `auth`. In this experiment an `if`-statement determines the program flow, simulating a situation where an authentication check is performed, as shown in Listing 2.

Table II
The success rates of the characterization experiments

	`unroll`		`auth`	
	Power	EM	Power	EM
ASILD1	87%	0.2%	60%	0.2%
ASILD2	0%	18%	N/A	57%
QM1	100%	N/A	N/A	N/A

```
flag = 1
...
trigger_up()
if (flag == 0):
    send_serial_authenticated()
else
    send_serial_denied()
trigger_down()
```

Listing 2. `auth` scenario pseudo code

A successful glitch affects the `if`-statement so the unintended branch – `send_serial_authenticated()` – is executed.

C. Results

These two experiments were prepared and performed on the three different targets over a period of three weeks. A total of ~720,000, ~550,000 and ~900,000 glitches were injected in the ASILD1, ASILD2 and QM1 respectively. The final results of the characterization part of this work are given in Table II. These percentages represent the rates of successful and undetected glitches obtained for the last FI campaign of each characterization experiment, after finding and using the most effective glitch parameters. Around 10,000 glitches were injected in each of these last FI campaigns. Other campaigns – not reflected in this table – were able to successfully affect the program execution but were detected by the MCU safety mechanisms.

For ASILD1, EM glitching was attempted only to verify its feasibility. The EM FI campaigns did not attempt to find the most EM sensitive point in the chip surface and the optimal parameters. Therefore only a 0.2% of success rate was achieved after one day of testing. If the optimal parameters are found and used, it is reasonable to expect a success rate similar to voltage glitching.

On ASILD2, voltage glitching proved to be ineffective in the `unroll` experiment. Furthermore, none of our glitches were reported as detected. We find a possible explanation in the internal power distribution and the on-chip voltage regulator that could be filtering the glitches. We did not try voltage glitching in the following ASILD2 experiments, limiting them to EM glitching. QM1 is also characterized, by using the unroll experiment and voltage glitching. Due to a high initial success rate, no more characterization experiments are run as similar results are expected in the rest of the tests.

Analyzing the information collected during the ASILD1 campaigns, we find that the most effective detection mechanism is the lockstep output comparison, which was triggered in 84% to 98% of the detected glitches, varying from experiment to experiment. Flash and RAM parity errors were triggered in 18% to 25% of the detected glitches. Other safety mechanisms were triggered in less than 3.5% of the detected glitches. Note that a glitch can trigger many detection mechanisms at the same time. The ASILD2 target does not report faults with enough detail to analyze the effectiveness of the different safety mechanisms.

Contrary to our initial expectations, all the previous described experiments succeeded while using a single glitch. Moreover, when using two glitches separated by the same number of clock cycles as the lockstep delay, the success rate decreases drastically.

The primary conclusion to be drawn from these characterization experiments is that it is possible to find glitch parameters for both ASILD1 and ASILD2 which result in a relatively high rate of successful glitches, without triggering any detection mechanism. Lacking a detailed insight into these mechanisms, it is hard to determine with certainty the cause of them failing to detect more glitch attempts. We suspect that the cause could be in the fact that both ASIL-D targets implement lockstep configurations that only compare the CPU output signals and not the internal CPU state. Both targets use a two-clock cycles delay lockstep with pipelined CPUs. This suggests that an injected glitch could affect the same instruction being executed in the two lockstep cores but in different stages of the pipeline, resulting in similar output signals and not triggering the lockstep error. Furthermore, the fact that only output signals are taken into account means that if faults are injected in memory shared between the cores – a part of the system that is not run in lockstep – both cores will execute the same faulty instructions or operate on the same faulty values. Error control schemes for these memories should offer a solution here, but they might be insufficient according these results.

V. Breaking JTAG protection

Previously discussed characterization experiments show that it is possible to use FI on ASIL-D MCUs to affect the code execution flow and bypass instructions, even if lockstep processing units and ECC protected memories are used. These experiments are run in a white-box manner in a controlled environment with full control over the software, including a trigger signal. Real attacks are typically executed in a black-box manner where the attacker has no control over the CPU and no access to the code. In order to have a better understanding of the risks that FI attacks pose on a real application, a new FI experiment that represents a non-controlled environment is prepared. As debug interface access is one of the first assets that an attacker tries to obtain, the resistance of the debug interface protection mechanisms present in our targets is assessed against FI attacks in this new experiment.

Most modern MCUs have a permanent protection mechanism for closing the debug interfaces. This protection is usually enabled at the end of the manufacturing process of the embedded system in order to protect the firmware contained in the MCU. It is a common practice for many semiconductor manufacturers to implement this debug interface protection in the ROM code executed at boot time, before the application firmware is executed. This ROM usually reads the protection configuration from an one-time programmable (OTP) memory section or a word in the internal flash and configures the registers of the debug interface module accordingly. An attacker using FI to bypass the configuration read or to disturb the debug module configuration moment would effectively unlock the debug interface on a protected device, which is considered here as successful.

Successful FI attacks rely on reliable trigger signals to ensure precise timing of a glitch. In the characterization experiments, the trigger signal was generated by our firmware. In this experiment, because the ROM is immutable, the boot process firmware cannot be modified to generate the trigger. Instead, side channel analysis (SCA) – more specifically, power analysis – is used. We observe that by comparing the power consumption traces of the boot process in a protected and an unprotected chip, an attacker can find the moment in time where the debug interface protection is enabled and get the timing information needed for injecting a glitch. Then, we use the icWaves tool to generate a trigger from the identified power consumption pattern. In this work this triggering technique and its results are presented for ASILD1 and ASILD2. For comparison purposes, this technique is also applied to the QM1 target.

Note that of all the debug interfaces supported by our targets – JTAG, UART, CAN – we choose attacking JTAG because it is the most common one. All descriptions of how specific targets implement their protection mechanisms below are based only on information found in publicly available documentation and resources.

A. ASILD1

The debugging interface in ASILD1 can be locked by programming a 128-bit password in the OTP area. By profiling power consumption at boot time both in the locked and unlocked state, and comparing the two, a small difference in the power consumed can be observed in a certain section, as shown in Figure 1. This profiling technique is commonly known as differential power analysis (DPA). The power consumption difference is attributed to the ROM reading the JTAG password and locking the debug interface. We run several FI campaigns around this point in time to prevent the JTAG lock.

~259,000 glitches were injected in total for a broad range of different glitch parameters. After tuning the glitch

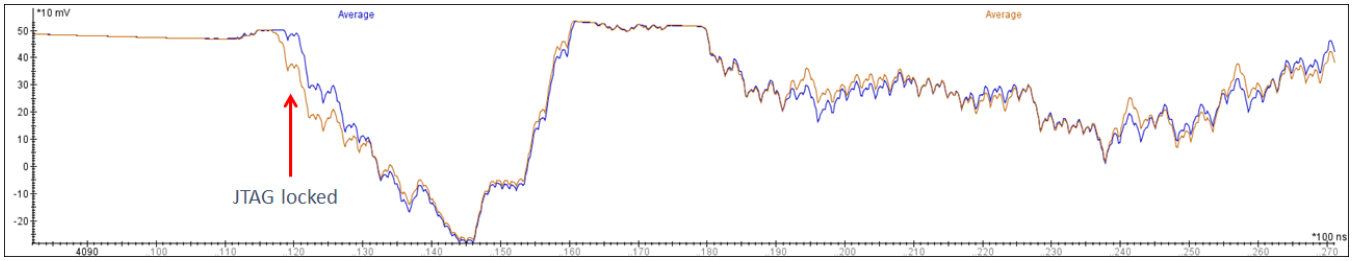

Figure 1. Difference between locked chip (brown) and unlocked chip (blue) for ASILD1

parameters, 35,200 glitches with the optimal parameters were injected. 1.3% of the attempts succeeded to keep the JTAG unprotected without triggering any lockstep or memory faults. An additional 1.2% of the glitches succeeded to keep the JTAG unprotected but were detected by the safety mechanisms. Finally, 3.2% of the attempts had no apparent effects other than triggering the fault detector mechanisms. This means that the ASILD1 target can be unprotected in less than one minute and half approximately, after performing the preparatory profiling and tuning steps.

B. ASILD2

ASILD2 has several configuration parameters that determine if its debugging interface is locked:

1) Life cycle state – indicates whether the product is still in production or deployed in the field.
2) Censorship state – blocks access to the debugging interface and flash memory.
3) Four debug lock bits – control access for the different JTAG clients.

The configuration of these parameters is stored in the OTP memory. The default value for those parameters that are not configured is "locked" except for the life cycle, which is unlocked by default and has to be configured by the implementor before releasing its final product. Only when all three of these configuration parameters are set to their locked state is the interface properly protected.

Immediately after the reset and before executing the ROM code, the configuration is pre-loaded by a state machine that reads it from OTP and loads it into the internal configuration registers. A second state machine verifies the written values. An attacker attempting to unprotect the JTAG has to interrupt the pre-load of the life cycle state or, if configured in the final product, the censorship or the debug lock bits.

The censorship and the debug lock bits are configured as entries or records in a register configuration table stored in OTP. Each record contains a register address and its value. The pre-load state machine iterates through the table and configures the registers. By comparing the power consumption of the boot process with different configuration tables, two points can be identified where the target activity is proportional to the number of records in the table. These points likely correspond to the pre-load and verification operations. Figure 2 and Figure 3 show the differences for the pre-load and the verification respectively when the table has 2, 102 and 120 records.

We prepared a target with the censorship and the debug lock bits configured and we ran a FI campaign to glitch the pre-load state machine when iterating the configuration table. As characterization attempts with voltage glitching were unsuccessful, only EM glitching is used for this experiment. After scanning the entire chip surface by injecting ~240,000 glitches, 0.34% of the glitches were successful in corrupting debug lock bits configuration. The censorship state was not affected by any glitch. Only a very small part of these glitches, <0.001%, was reported by the module responsible for transferring the records as configuration errors. No more FI campaigns trying to optimize the glitch parameters were ran. The success rate would be higher if optimal parameters where used, especially the position in the chip surface. It is not clear to us why the censorship state was not affected by the glitches and why the verification state machine did not detect more errors.

The life cycle state is not configured using the register configuration table but by writing the value in an OTP register which is directly read by the pre-load state machine. Being able to successfully influence the life cycle state that is configured during boot offers a much more powerful attack than by affecting the censorship or the debug lock bits, as it is not hindered by the circumstantial configuration requirements needed to attack these other parameters.

By comparing the power consumption of the same sample with different life cycle states, differences can be observed in many points during the boot. All these points are potential candidates for glitching to prevent the configuration of the life cycle. We choose to run a FI campaign around the candidate that is closer in time to the spot where we identified that the configuration table was being read. After injecting ~500,000 glitches, we did not succeed in affecting the life cycle configuration. Further FI campaigns with the other candidates are required to fully discard the possibility of attacking the pre-load of the life cycle.

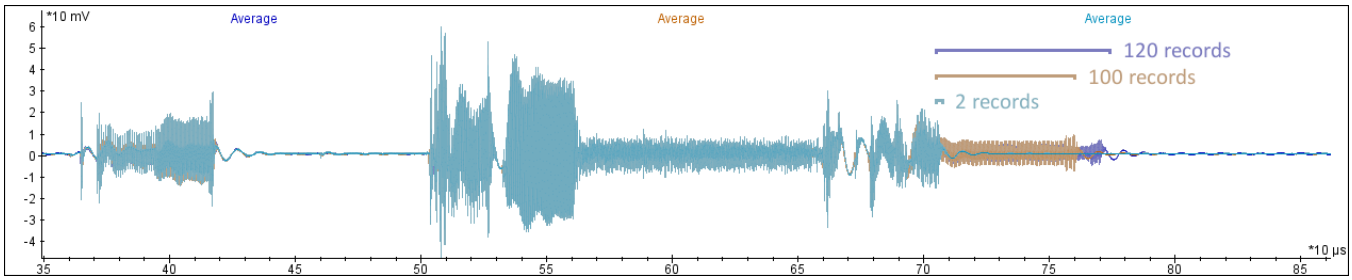

Figure 2. Difference between 2 (cyan), 102 (brown) and 120 records (blue) programmed for ASILD2, point one: load values

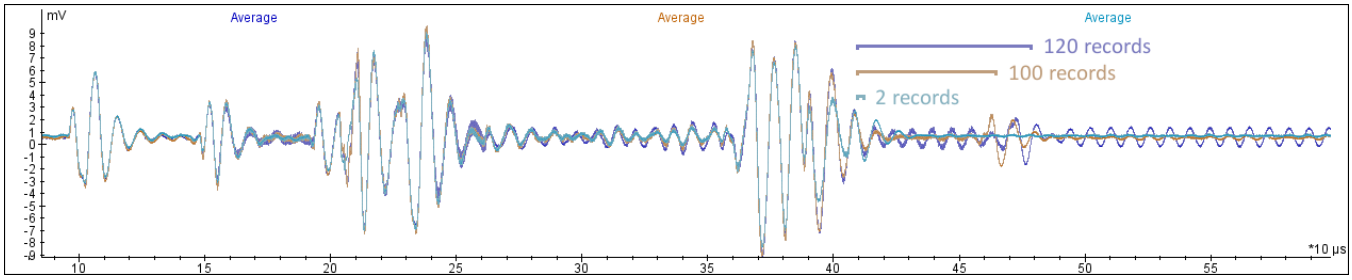

Figure 3. Difference between 2 (cyan), 102 (brown) and 120 records (blue) programmed for ASILD2, point two: verify values

C. QM1

For comparison purposes, the JTAG port of the QM1 chip is also tested. The state of the debug interface is defined by a 32-bit word stored in internal flash, which is read and used by the boot ROM to lock the JTAG port. Because the JTAG configuration word is stored in flash and not in OTP, it can be rewritten multiple times to run a correlation power analysis (CPA). By randomly changing the JTAG configuration word and correlating its Hamming weight with the power consumption, the moment where the ROM reads the configuration word can be find.

Several voltage FI campaigns are ran to alter the JTAG configuration value read from flash. ~89,000 glitches are injected in total. In the last campaign, after finding the optimal glitch parameters, 80% of the ~10,000 attempts succeeded in unprotecting the JTAG interface, which means that the attack take less than 5 seconds to succeed.

D. JTAG experiments conclusion

The experiments succeeded in unlocking the JTAG interfaces on the ASILD1 and the QM1 targets, and in the ASILD2 target only when the debug lock bits are configured. Further investigation is needed in order to determine whether the ASILD2 chip can be attacked in other scenarios. It is not possible to directly compare results on Table III because the three chips are implemented with different technologies which could have different sensitivity to FI attacks. Moreover, the experiment on the ASILD2 chip did not attempt to find the optimal glitching parameters to maximize the success rate. In any case, the big gap between success rates between the ASIL and non-ASIL chips indicates that some of the ASIL-D safety mechanisms could slightly mitigate the FI attacks, but cannot fully prevent them.

Table III
AN OVERVIEW OF THE SUCCESS RATES OF THE DIFFERENT JTAG EXPERIMENTS

password ASILD1	lock bits ASILD2	life cycle ASILD2	config QM1
1.3%	0.34%	0%	80%

VI. SECURITY CONSEQUENCES

The experiments described in Section IV and Section V prove that FI attacks can bypass the detection mechanisms outlined above in state of the art ASIL-D certified MCUs. Although the safety mechanisms detect some faults, they cannot adequately prevent a FI attack.

The fact that it is possible to bypass instructions or execute an unintended branch means that the integrity of the code execution cannot be assured. Any security feature enabled and any check executed in the software can be manipulated by the attacker as an important step toward gaining access to assets. Various assets can be targeted, depending on the attacker profile and the attack scenario. The most common assets include firmware extraction, firmware manipulation, runtime control and access to debug interfaces.

The firmware contained in MCUs is usually the most valuable asset to Tier1 manufacturers and OEMs. The firmware implements those functionalities that add value to the product and differentiate the brand from its competitors. Taking for example a brake controller assistant or an autonomous

driving system, there are thousands of hours invested in research and development of the perfect algorithms for stopping the car safely or driving it autonomously through the city. A competitor extracting the software of an ECU would have access to those proprietary technologies and could copy them, or use them to improve their own technology and gain an unfair competitive edge. Extracting the firmware can also be an intermediate step for achieving a bigger objective from a security point of view. An attacker could reverse engineer the extracted software to understand its operation and find a vulnerability. This has been done several times in the past when researchers [8], [9], [10] reversed the firmware of the remote key entry systems and immobilizers. Vulnerabilities in the cryptographic algorithms were found, which allowed cloning the keys to open and start the car. Similarly, researchers [11] reversed the extracted software to find remotely exploitable vulnerabilities which gave control of the car without even needing physical access to it.

Modifying the firmware is another common objective of attackers. Typically this attacker profile includes car tuners who seek to modify the performance of their vehicles by altering the software. These persons normally are not aware of the risk that those modifications could pose to theirs and others lives. A less common profile could be subjects that are purposely looking to inflict damage in the vehicle or to its occupants by the means of sabotage. In both cases, the possibility of altering the MCU firmware without any tamper evidence is appealing and it should be prevented by the manufacturers. Extracting and modifying the firmware can be done, for example, after bypassing the authentication scheme of the Unified Diagnostic Services (UDS) protocol, if present in the targeted ECU. The authentication experiment described in Section IV resembles an attack on the UDS authentication.

Runtime control and access to debug interfaces are usually intermediate steps for gaining any of the other assets mentioned. For example, the attack executed in Section V for gaining JTAG access could be used to extract and modify the firmware in the MCU. An additional benefit of gaining runtime control or debug access is the possibility of experimenting and debugging a live target, which could help to reverse and understand the entire system. Runtime control can be gained using FI through the means of accessing the debug interface, bypassing the Secure Boot or using more advanced techniques like manipulating the CPU's program counter [4].

Finally, the goal of an FI attack could be simply to disable a security feature. For example, this could be the case for the immobilizer that verifies the response of the key in order to allow starting the engine. By using FI, the attacker could bypass the verification check and start the car without the legitimate key.

VII. Recommendations

As described before, safety mechanisms implemented in ASIL-D chips can reduce the success rate of FI attacks but not fully prevent them. In the absence of other hardware countermeasures against fault injection, software mitigations like execution flow control, redundancy or random delays should be implemented. The reader can refer to the extensive documentation (e.g. [12]) detailing these countermeasures.

It is not only responsibility of the embedded system developers (e.g. Tier1 manufacturers) to implement these countermeasures in their software. As explained in Section V, ROM code can also be the target of FI attacks and therefore chip manufacturers must also implement countermeasures when developing ROM code.

When using a MCU with a vulnerable ROM, embedded system developers can mitigate the risk by using the ROM patching mechanism, if such a mechanism is present. For example, ARM Cortex MCUs usually implement the Flash Patch and Breakpoint Unit (FPB) which can be used to patch ROM code. Depending on the MCU manufacturer, the FPB can be enabled at reset time and patch the boot routine, or can only be enabled after booting and patch only the ROM drivers used in the application. In the experiment performed on ASILD1 described in Section V, 1.2% of the glitching attempts succeeded to unprotect the debug interface protection mechanism, but lockstep or memory errors were detected. The application developers should check for these errors or any other unexpected state (e.g. unexpected values in a JTAG configuration register) immediately after booting and reset the chip if any inconsistency is found. In the specific case of the ASILD2 target, adding redundant records to configure the censorship and the debug lock bits could mitigate an FI attack on them.

VIII. Future work

This work leaves several open questions that are interesting for future work:

- Perform similar investigations on additional targets that implement these safety mechanisms to further identify their effectiveness as countermeasures.
- Apply fault injection attacks on real world systems using ASIL-D MCUs (i.e. ECUs) to confirm that these attacks pose a real threat. It is especially interesting to test implementations of the UDS authentication scheme.
- Test targets that allow disabling and tuning the safety mechanisms individually and characterize the effectiveness of each mechanism. For example, all targets tested in this work use a two-clock cycles delay between the two lockstep cores. Different delays might be more or less effective.
- Do additional characterization experiments with the presence of software countermeasures. This should give

an insight in what type of setup would be better in combating fault injection attacks.

IX. Conclusion

We assessed the resistance of two ASIL-D MCUs against fault injection attacks using both voltage and EM glitching techniques. Both targets appeared highly sensitive to at least one of these techniques in our characterization experiments, showing that basic conditional branch type code structures can successfully be attacked in ASIL-D targets. Furthermore, we succeeded in identifying and attacking the JTAG protection by using power analysis and voltage glitching in one ASIL-D and one QM target and extracted secret information from its firmware. Limited success was obtained attacking the JTAG protection of the second ASIL-D target. Each experiment required approximately one week of preparation, but once the attacker finds the optimal glitching parameters, the attack can be repeated in less than an hour, including the time needed to modify the application PCB.

We observed that, compared to standard non-ASIL MCUs, the safety mechanisms implemented in ASIL-D chips can help to mitigate fault injection attacks, but not fully prevent them, nor can they reduce the risk that these attacks pose to an acceptable level. We advise MCU manufacturers and embedded system developers to implement both hardware and software countermeasures against fault injection.

References

[1] ISO26262, "ISO 26262, Road vehicles — Functional safety," https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en:term:1.134.

[2] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, Feb. 2006.

[3] P. Tummeltshammer and A. Steininger, "Power supply induced common cause faults-experimental assessment of potential countermeasures," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, Jun. 2009, pp. 449–457.

[4] N. Timmers, A. Spruyt, and M. Witteman, "Controlling PC on ARM Using Fault Injection," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2016 Workshop On*. IEEE, 2016, pp. 25–35.

[5] I. Biehl, B. Meyer, and V. Müller, "Differential fault attacks on elliptic curve cryptosystems," in *Annual International Cryptology Conference*. Springer, 2000, pp. 131–146.

[6] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of eliminating errors in cryptographic computations," *Journal of cryptology*, vol. 14, no. 2, pp. 101–119, 2001.

[7] C. Giraud, "DFA on AES," in *International Conference on Advanced Encryption Standard*. Springer, 2004, pp. 27–41.

[8] F. D. Garcia, D. Oswald, T. Kasper, and P. Pavlidès, "Lock It and Still Lose It–On the (In) Security of Automotive Remote Keyless Entry Systems," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[9] M. Kasper, T. Kasper, A. Moradi, and C. Paar, "Breaking KeeLoq in a flash: On extracting keys at lightning speed," in *International Conference on Cryptology in Africa*. Springer, 2009, pp. 403–420.

[10] R. Verdult, F. D. Garcia, and B. Ege, "Dismantling megamos crypto: Wirelessly lockpicking a vehicle Immobilizer," in *Supplement to the 22nd USENIX Security Symposium (USENIX Security 13)*, 2015, pp. 703–718.

[11] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, vol. 2015, 2015.

[12] I. Verbauwhede, D. Karaklajic, and J.-M. Schmidt, "The fault attack jungle-a classification model to guide you," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop On*. IEEE, 2011, pp. 3–8.