ElectroMagnetic Fault Injection Characterization

George Thessalonikefs
george.thessalonikefs@os3.nl

University of Amsterdam
System & Network Engineering MSc

February 10, 2014

Abstract

This paper tries to characterize the effects of Electro Magnetic Fault Injection in embedded devices. Transient electromagnetic pulses in the surface of an ARM processor are used in order to induce faults in the execution of software. Data-processing and Memory instructions are the targets of the attack. Faults can be found during the fetch, decode, write-back phases.

Contents

1 Introduction

Fault injection attacks are proven to be practical and pose a risk against the secure operation of embedded devices. Contrary to Side Channel Attacks where the side channels (power consumption, electro magnetic radiation, etc) of an Integrated Circuit (IC) are observed in order to reveal information, Fault Injection attacks try to have an active impact on the IC's operation by skipping/corrupting security operations, corrupting registers and in general perturbating the IC's core operations. They are often used to bypass protection measures such as PIN verification or even to extract secret information (eg. private keys). Examples of such attacks can be found in [1], [2] and [3]. Unfortunately, reasoning what the injected fault causes inside the chip is very difficult and usually only the result is clearly describable (e.g. successful bypass of a security feature).

Common methods of fault injection into an Integrated Circuit (IC) such as smart cards or embedded devices are the following:

- **Voltage/Clock fault injection**, by introducing dips or spikes in the Vcc/Clock line of the target
- **Optical fault injection**, by targeting certain areas of the IC with a laser
- **Temperature fault injection**, by heating/cooling the IC outside of its thermal tolerance range
- **Electromagnetic fault injection**, by using a magnetic field close to the IC.

Voltage/Clock and Optical fault injection techniques are the most common and succesfull attacks against ICs. They are also the most countered ones by using glitch sensors and light sensors respectively for example. Glitch sensors may not allow the voltage in the supply/clock line to exceed a certain range and light sensors may destroy the private data if optical fault injection is detected. Both these fault injection techniques require preparation for the target, leaving evidence of intrusion, in the form of isolating the power/clock lines in the case of voltage/clock fault injection or decapsulating the chip in the case of optical fault injection.

Electro Magnetic Fault Injection (EMFI) can bypass the aforementioned countermeasures and by its nature is harder to detect during run-time, leaving little or no evidence. At the time of writing and to the author's knowledge there are no specific countermeasures for EMFI. Generic countermeasures during software implementation can be used (eg. always validating input, state of registers, etc) in order to make fault injections in general harder to accomplish but not impossible.

Temperature fault injection is the hardest to achieve and control because of the exact timing needed between the target operations and the temperature variations that are to take place.

As the title of the paper implies, the research will focus only on EMFI. Previous research [6], discusses how EMFI works, confirms its feasibility on smart cards and embedded devices and presents the correlation between different glitching parameters (type of coil, distance to the target, etc) and the occurrence of faulty behavior.

1.1 Research question

Electro Magnetic Fault Injection (EMFI) is quite recent and although already successfully used to attack software implementations of the Chinese Remainder Theorem(CRT)-based RSA algorithm as shown on [1], or perturbate the output of a True Random Number Generator(TRNG) as shown on [2], little can be said about the state of the chip during the injection phase. The aim of this research is to try and give an answer to the following question:

What are the effects of Electro Magnetic Fault Injection (EMFI) on embedded chips?

Previous research [4], focused on understanding the effects of voltage fault injection on a specific target. This research will compare the effects of EMFI with voltage fault injection where possible.

1.2 Scope

Due to the research's limited time of 4 weeks the scope of the research has to be carefully defined. The target chip of the research will be Freescale's i.MX6S, a single core applications processor built around the ARM Cortex-A9 IP core. Riscure's own Electro Magnetic Fault Injection Transient Probe will be used to conduct the experiments.

Inducing glitches into the target is expected to yield a multitude of results indicating the faulty operation of the target. Only the ones that we can reason about based on the experiment's parameters are to be discussed.

EMFI may follow various principles: magnetic transient pulses or harmonic electromagnetic injection. Magnetic transient pulses induce a voltage glitch in any circuit loop under the coil, which may change a transistor status instantaneously from 'OFF' to 'ON', or vice versa depending of the polarity of the voltage glitch and type of transistor, and thus can target specific sensitive operations[5].

Harmonic electromagnetic injection, on the other hand, with constant injection of EM waves in a set frequency, targets the entire operation of parts of the chip such as ring oscillators used for True Random Number Generators [2].

Harmonic electromagnetic injection is out of the scope of this research.

2 Approach

In order to study the effects of EMFI several research parameters have to be defined. Namely, the EMFI hardware setup required, the glitching parameters used with the EMFI setup, the target chip that the tests will be run on, the executable code that we will try to glitch and lastly the timing parameters to be able to target specific areas in that code.

2.1 Glitching parameters

A number of glitching parameters will be used in order to define the glitch we are trying to produce. Namely:

Type of coil
The option of using 1.5mm or 4mm diameter coil was present. The 4mm diameter coil was used. As shown in previous research [6], "the 1.5mm diameter coil did not produce any glitches. This is due to the thicker encapsulation of the chip, which influences the distance between the coil and the target".

Two-dimensional coordinates
The whole area of the chip was the input for the initial scans. Based on the results, more precise coordinates were given to investigate specific areas of the chip. The orientation of the chip was such that the corner where pin 1 resides (marked by a dot on top of this chip) is presented with coordinates of (0, Ymax) in the following diagrams.

Glitch source power (hardware dependent)
The power supplied to the coil. It influences the power and range of the produced magnetic field. Varying values will be set during the tests in order to gather as much diverse data as possible.

Glitch duration (hardware dependent)
The duration, in nanoseconds, where power is provided to the coil. Varying values will be set during the tests in order to gather as much diverse data as possible. The minimum value that could be used and result in current running through the coil, measured with oscilloscope, was 12 nanoseconds.

Glitch offset
The time offset, in nanoseconds, after a trigger that a glitch is to take place. Varying values will be set during the tests in order to gather as much diverse data as possible.

2.2 Target

The target of the research is the 32-bit ARM Cortex-A9 processor which implements the ARMv7-A architecture based on the RISC architecture. The Cortex-A9, being one of the state of the art processors used in smartphones, tablets, home media players, etc, has many advanced features (such as floating point processing engine) that will not be used during this research. Thus, features of the Cortex-A9 processor relative to the research include:

Clock speed
The Cortex-A9 was used with a clock speed of 792 MHz. This results in approximately 1,26 nanoseconds per clock cycle.

Pipeline
The ARM Cortex-A9 has a double-issue superscalar, out-of-order, speculating 8-stage pipeline.

Double-issue superscalar refers to the fact that in every cycle two instructions can be inserted in the pipeline simultaneously.

Out-of-order refers to the fact that during execution the processor can choose to execute non-sequential but data-independant instructions in order to improve its efficiency.

Speculating refers to the fact that the pipeline implements a branch prediction mechanism in order to improve its efficiency. When a branch instruction is encountered, the processor decides if the branch is likely to be taken or not, based on the branch prediction mechanism, and inserts the respective instructions into the pipeline. If the condition of the branch was mispredicted, the now invalid instructions already in the pipeline are dropped, replaced by No Operation(`NOP`) instructions, introducing a delay until the correct instructions enter the pipeline.

Registers
The following 16 x 32-bit registers available in the Cortex-A9 were used.
Registers `R0-R12` are general purpose registers.
Register `R13`, also referred as `SP`, is the Stack Pointer.
Register `R14`, also referred as `LR`, is the Link Register.
Register `R15`, also referred as `PC`, is the Program Counter.

Barrel Shifter
It provides a mechanism to carry out shift operations. It can be used in conjunction with other instructions to provide shift operations on the second operant (immediate value or register) at the cost of execution time.

Instruction execution times
All instructions used during the various tests of the research are data-processing instructions and specifically: `MOV` (move), `MVN` (move negative), `LSL` (Left Shift Logical), `EOR` (Exclusive OR), `BIC` (Bit Clear). The barrel shifter was not used in conjunction with any instruction resulting in 1 cycle/instruction execution time for the aforementioned instructions.

2.3 Code Instrumentation
In order to be able to observe and reason about glitches happening during the tests, a comparison between the correct and the faulty output is required. For this purpose a test program was written in ARM assembly. Assembly was used a) in order to escape the optimization offered by the C's compiler that may change the execution order of the code, the registers used, etc, and b) in order to have an as close as possible view of the actual instructions executed in the processor.

The instrumentation is the following:

1. Register initialization
2. Trigger ON
3. Critical Code
4. Trigger OFF
5. Send the data

Register initialization
`R4-R11` will not be used and will be set to known values.

R0 and R1 will be used during the critical code phase and will be assigned 0xFFFFFFFF and 0x01 respectively.
R2 and R3 will be used for the Trigger ON/OFF phases and will be assigned specific values according to the GPIO specifications of the board. R3 will contain the address in memory where a specific bank of pins is mapped and R2 will contain the value to assign to the target pin.
R12-R15 are crucial to the processor's state and execution and will be set by the processor. They are set in a deterministic way and they are expected to have the same values in each execution of the code.

Trigger ON/OFF
During this phase a certain GPIO pin will be altered to inform about the entrance/exit to the critical code phase.

Send the data
As a last step, data will be sent through serial communication from the target. The data will contain the aforementioned registers and their respective values.

Critical Code
This is the part of the code that we are trying to glitch and observe the results. It uses the R0 and R1 registers that were initialized earlier. An unrolled loop of roughly 32 pairs of instructions resides in this area. Each instruction pair consists of a logical operation followed by a shift operation.

The logical operation is a BIC instruction between R0 and R1. BIC (Bit Clear) is a synonym for an AND NOT logical operation. The result of the BIC instruction between two operands is to clear the bits (flip to '0') of the first operand that in the second operand are represented by '1'. As an example if the values of the first and second operand are 0b01101 and 0b11000 respectively, the result would be 0b00101.

The shift operation is a LSL (Left Shift Logical) instruction between R1 and the immediate value 1. This will result in R1 to be shifted 1-bit to the left, filling the rightmost bit with '0'.

As an example, an illustration follows of the first three executions of these instruction pairs. The values are represented in binary for simplicity and only the 4-least significant bits are presented:

```
1.    R0: 1111    R1: 0001
2.    R0: 1110    R1: 0010
3.    R0: 1100    R1: 0100
```

At the end of the execution we expect the following correct output (registers contain whole words now and are represented in hexadecimal notation):

```
R0: 00000000      R1: 80000000
```

If a BIC instruction is not executed correctly we expect to see a non-'0' bit somewhere in the value of R0. Based on that bit we can deduce the time the glitch happened, meaning which specific instruction was hit.

If a glitch happens during an LSL instruction it is impossible to reason about the specific LSL instruction that was glitched. But given the glitch offset known from a nearby glitched BIC instruction we can reason that the previous or the next LSL instruction relative to that BIC instruction was glitched.

A second version of the same code was used. The only difference between the two versions is that the `BIC` instruction was replaced by an `EOR` (Exclusive OR) instruction. The expected final result remains the same. This second version was used for verification purposes. From now on we will refer to the two versions as `BIC` version and `EOR` version respectively.

2.4 Setup

To perform the tests during the research, an EMFI setup was composed and used. This setup is illustrated in a schematic in Figure 2.1. A picture of the setup can be seen in Figure 2.2. Components of the setup include:

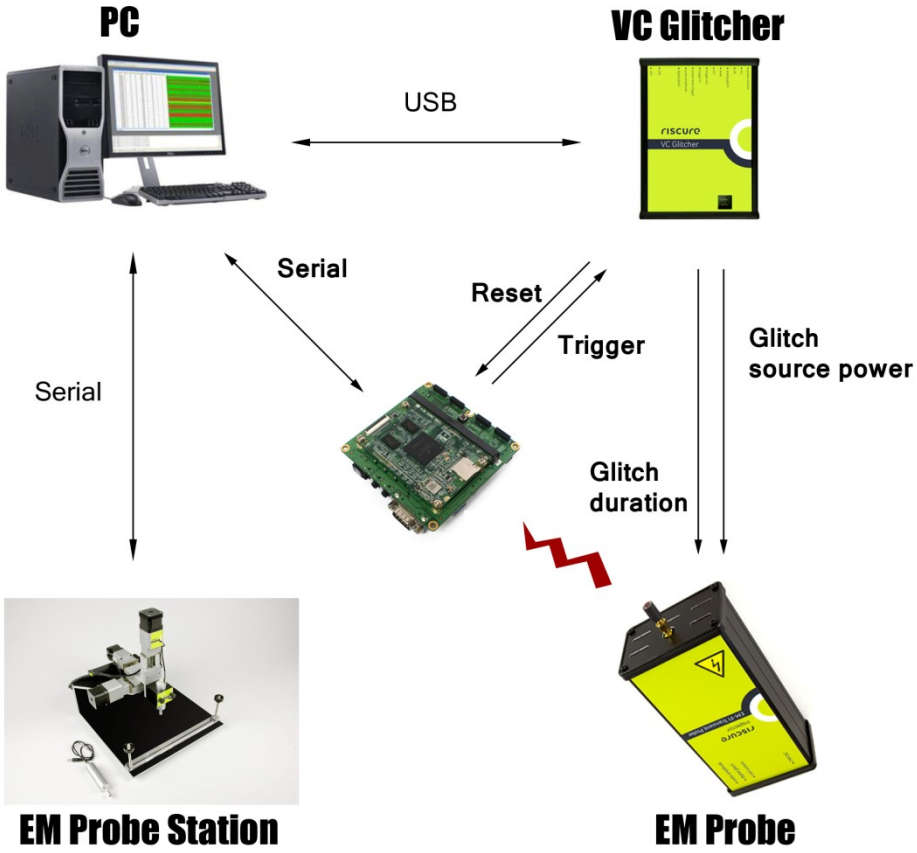

Figure 2.1: Schematic overview of the test setup

PC

Riscure's proprietary software, Inspector, is used to orchestrate all the other devices of the setup and record the results. It communicates through serial with the EM Probe Station in order to pass/receive coordinates, and with the target in order to issue the command to execute the test code and receive the result. It communicates through USB with the VC Glitcher in order to pass the glitching parameters.

Riscure's EM Probe Station

It offers XY movement to the attached device. The attached device in this setup is Riscure's EM Probe. The target is securely placed at the base of the EM Probe Station. The EM Probe Station offers a step size of 2.5µm and repetition error smaller than 50 µm to allow test repeatability [9].

Riscure's VC Glitcher
It is the main control unit during the tests. After the glitch parameters has been passed from Inspector it is responsible to generate the configurable glitch by passing the glitch duration and glitch source power parameters to the EM Probe when receiving a trigger signal from the target. In case the target becomes unresponsive, it sends a signal to the reset line of the target.

Riscure's EM Probe
It generates the electromagnetic field by passing current to the coil attached to its end. The maximum voltage over coil is 450V (+/- 10%) and the maximum current through coil of probe tip (4mm tip) is 48A (+/- 10%). The EM pulse power can be controlled for 5-100% of total power. For reaching full power, a pulse width of at least 50ns is required [5].

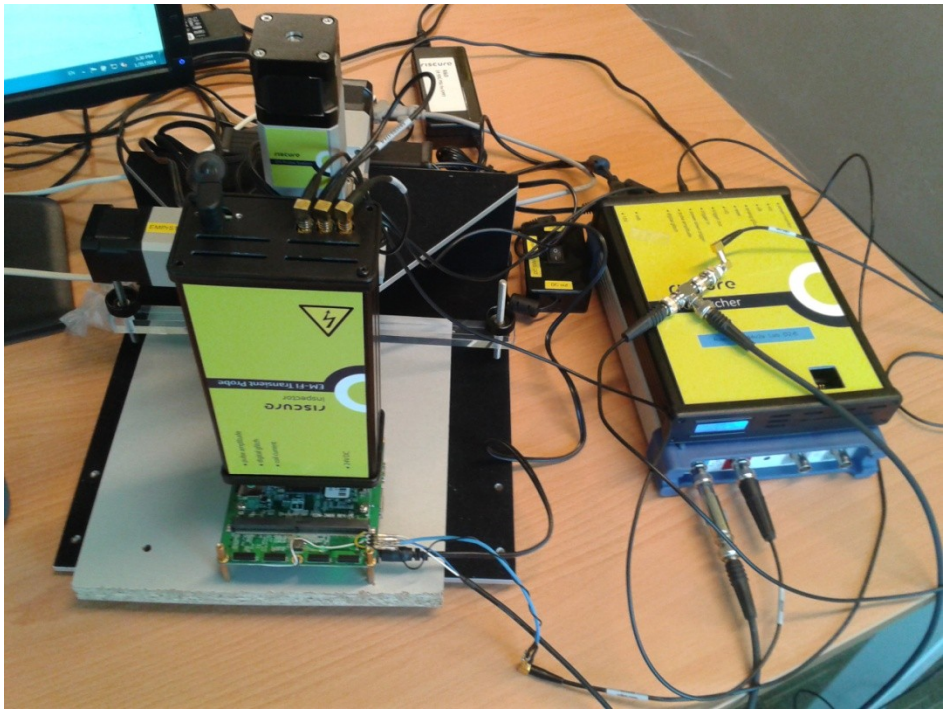

Figure 2.2: The EMFI setup

2.5 Note on glitch timing
As seen on Figure 2.3, there was a delay between the time of the Trigger ON event and the time the glitch was instructed to take place (glitch offset set at 0ns) measured at around 90ns. To compensate for this, several `NOP` commands were inserted between the Trigger ON code and the Critical Code in order to delay the Critical Code's execution by at least 90ns.

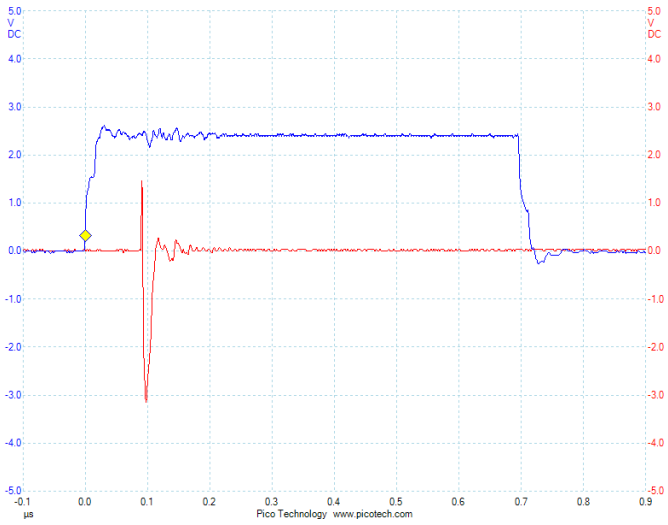

Figure 2.3: Trigger signal(blue), Coil current(red)

2.6 Glitch types

At this point we will distinguish the types of glitches we are going to get in 3 major categories:

Preferred glitches. The result follows the correct format (all the registers showing their values) but with different values than the ones expected in the registers. This type of glitches make it possible to reason what may have happened during the fault injection.

General glitches. The result is something unexpected. In this category, among other results, we expect to see exceptions thrown by the processor.

Mutes. They indicate that the target is not responding after the fault injection. This type of glitches offer no insight on what may have happened and the only action left is to reset the target and continue with the test.

3 Scans of the chip

The first step is to scan the chip for areas susceptible to EMFI. Not the whole area is expected to react to the magnetic field and of course not the whole area is expected to contain the semiconducting material(die). Every response, or lack of, that was different from the expected correct response is regarded as a glitch and is represented in the following heat maps except where otherwise stated. At the end of the scanning procedure we identify an interesting area of the chip where Preferred glitches are more likely to take place. Further tests will be focused on that area.

3.1 Full area scan

The first scan is a scan of the whole area of the chip. The result is shown on Figure 3.1.

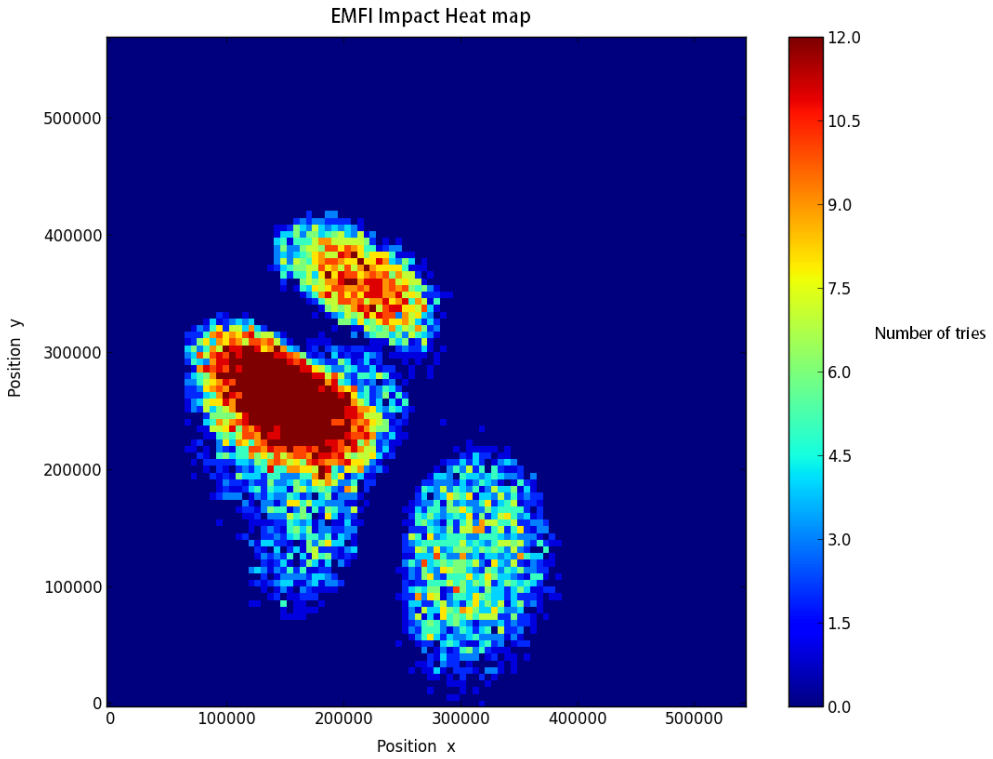

Figure 3.1: Full area scan heat map

Each colored square on the heat map represents a distinct region that was checked 12 times with different glitching parameters as mentioned in section 2.1. The color of the square represents how many glitches were encountered in total out of 12 tries. Three distinct areas are clearly standing out from the map. What caught our attention in this heat map is that the center of the chip, where we expect the die to reside, seems unresponsive. Further scaning of that area will be discussed in the next section.

If we filter the results and keep only the Preferred Glitches the heat map in Figure 3.2 reflects the results.

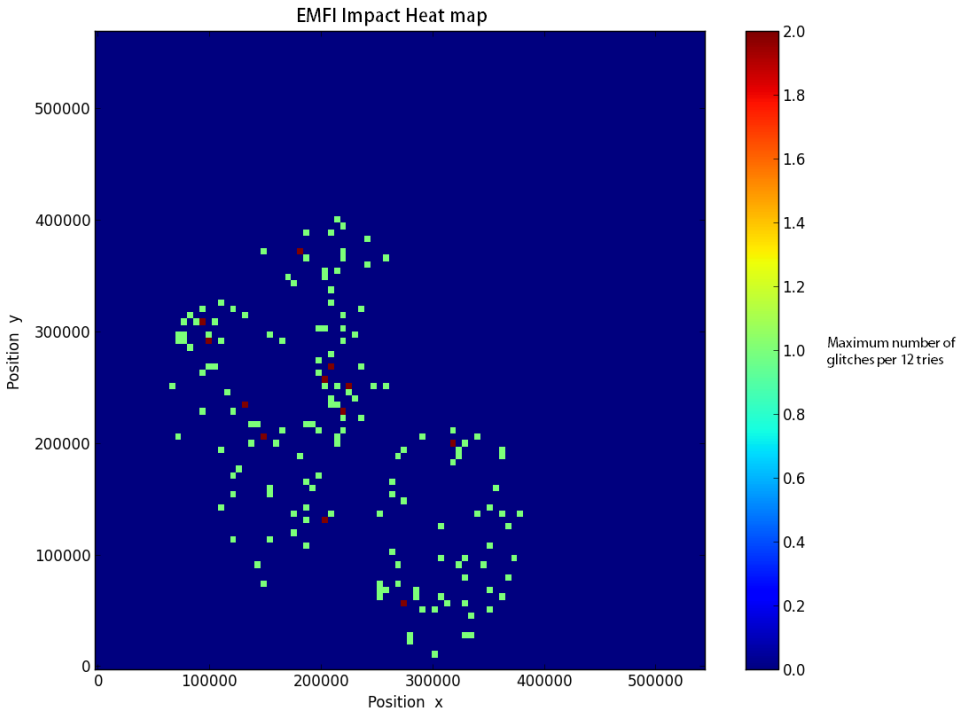

Figure 3.2: Filtered full area scan heat map

3.2 Scan of the die area

Our next approach is to scan the area directly above the die with more power to the coil in order to see if we can get any glitches. In order to roughly aim this area and be certain that is indeed located in the center, an ARM Cortex-A9 decapsulated chip was used as reference. The decapsulated chip is shown in Figure 3.3.

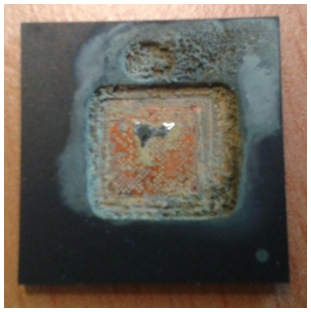

Figure 3.3: Decapsulated ARM Cortex-A9

The scan results can be seen in Figure 3.4.

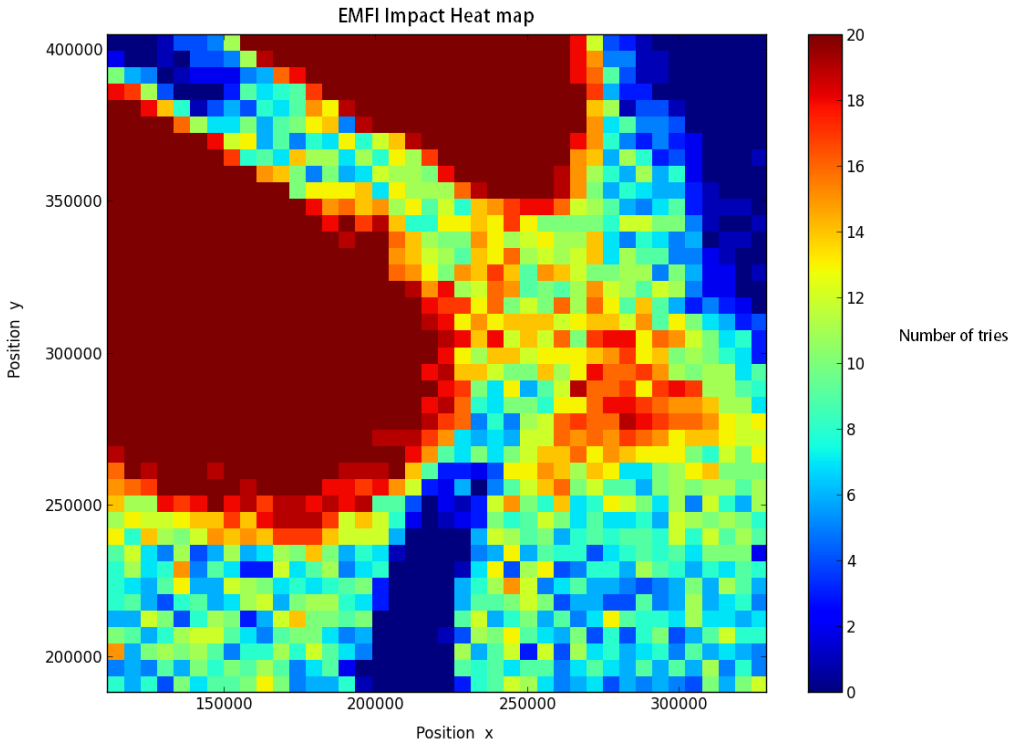

Figure 3.4: Scan of the die area heat map

It should be noted that the distinct deep red areas, though they seem interesting on the heat map they are mostly producing Mutes. An attempt to scan them with lower power produced Mutes or no glitches at all.

If we again filter the glitches for the Preferred glitches we get the following result shown on Figure 3.5. A different diagram other than heat map was used to better distinguish the location on the die.

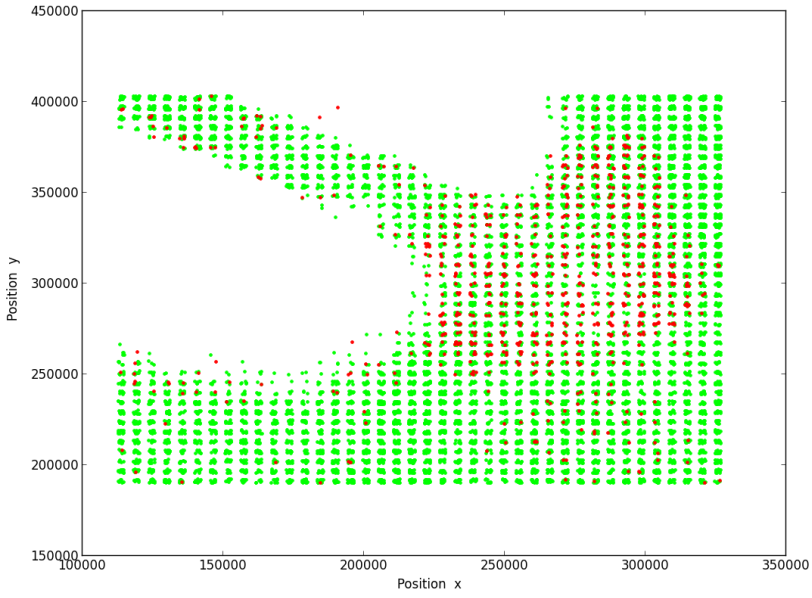

Figure 3.5: Area of Preferred glitches inside the die

In Figure 3.5 we see the area where Preferred glitches are present with red dots.

The green dots represent areas where no glitch was encountered. They are present in the diagram in order to give an outline of the scanned area and provide a distinction between the red and white dots.

The white dots (remaining white space) represent mostly Mutes and some General glitches.

3.3 Conclusion

From the consecutive scans of the chip we can come to the following conclusions:

- The outer edges of the die are more sensitive than the center of the die. More voltage was required in the coil in order to get glitches from the die area (Figure 3.4).
- Preferred glitches can be found in multiple areas of the chip and not exclusively in the center of the die as shown on Figure 3.2. This could prove useful in case the die cannot be targeted directly or the magnetic field cannot reach the semiconducting material due to extra layers of package atop the die.
- The red area depicted in Figure 3.5 is the area inside the die that Preferred glitches are likely to take place. This will be the main target area for further tests. Results will be gathered and examined mainly from this area.

4 Examination of glitch cases

In this chapter we will examine several glitch cases we encountered and try to reason what may have happened during the fault injection. The glitches mainly result in instuction skipping, reset issued by the processor and corruption of a related register.

4.1 Prefetch/Data abort exceptions

Prefetch abort exception is thrown by the Memory Management Unit when an instruction is going to be loaded from non-existing memory regions, meaning the memory address we are trying to fetch a command from is outside of the legal memory regions.

Data abort exception is thrown by the Memory Management Unit when an instruction is going to access the memory through an unaligned address.

These types of exceptions can be due to glitching the PC register directly, or by corrupting the stack, so that it points out of memory. In case of stack corruption, the PC register is also corrupted if the corrupted value is loaded back to the PC register.

This type of glitch resulted in a reset issued by the processor.

4.2 Undefined instruction exception

Undefined instruction exception is encountered when the result of decoding an instruction fetched from memory does not match any legal instruction.

Cases where the glitch might have taken place are during the fetch stage, the value retrieved from memory got corrupted, or the decode stage, where the result of decoding an instruction was corrupted.

With the presence and encounter of this type of exception we can reason that instructions that have reached the execution stage are likely to not be undefined instructions prior to the execution stage.

This type of glitch resulted in a reset issued by the processor.

4.3 Logical/Shift operations skipped

In this case we see that logical and/or shift instructions were skipped. Selected subcases follow. Only `R0` and `R1` are shown as the other registers all have the expected values. In these examples a `BIC/EOR` instruction was skipped. According to the undefined instruction exception above, this can be due to a glitch in the execution/write-back stage where the instruction was either not executed or the result of the instruction was not written back. The expected correct output will be presented first for comparison. The values are in hexadecimal notation.

- Correct Output
  `R0: 00000000  R1: 80000000`

- Corrupted Output - The first `BIC/EOR` instruction was skipped
  `R0: 00000001  R1: 80000000`

- Corrupted Output - An `LSL` instruction was skipped
  `R0: 80000000  R1: 40000000`

  It should be noted that, as mentioned in section 2.3, we cannot know which specific `LSL` instruction was skipped from this result alone. We have to verify it by comparing the glitch offset used for this glitch with glitch offsets used for the previous subcase glitches. If the glitch offsets match we can reason that these two instructions are neighbors.

- Corrupted Output – BIC/EOR and LSL skipped
  `R0: 80000001  R1: 40000000`

- Corrupted Output – EOR skipped – glitched write-back
  `R0: FFFFFFFE  R1: 80000000`

  This is a special subcase that we can detect only with the use of the `EOR` version of the code. By looking at the value of `R0`, we can see that something probably went wrong in the first `EOR` instruction as `0xE` = `0b1110`. We can reason that in the case of a write-back fault all the bits of `R0` flipped to '`0`'. In this case consecutive `EOR` instructions between '`0`' and '`1`' (from `R1`) will yield '`1`' as the result, thus the aforementioned `R0` value.
  In the case of the `BIC` version of the code, the result of `R0` under the same circumstances would remain `R0:00000000` thus making it impossible to detect such fault.

4.4 Comparison with Voltage glitching

Comparing our results with the ones from the research in voltage fault injection [4] several similarities and differences can be found.

Instruction skipping was feasible in both cases. However, in voltage glitching the instruction fetch stage was likely to be glitched whereas in EMFI the execution/write-back stages were the main suspects. The XMEGA used in [4] was handling illegal/undefined instructions as `NOP` instructions, essentially skipping the targeted instruction. In such case, the ARM processor was throwing exceptions and a reset was taking place.

Unrelated register corruption, meaning changes in the values of registers not related to the instructions under test, was observed during voltage fault injection. No such cases were observed in the Preferred glitches during EMFI.

During various tests with voltage glitching a tendency for bits to transition from '`1`' to '`0`' was observed. The same was observed with EMFI in the write-back phase as discussed in section 4.3.

5 Conclusion

During this research the effects of Electro Magnetic Fault Injection on embedded devices were observed. The target of the research was the 32-bit ARM Cortex-A9 processor. A test code was written in ARM assembly and run on the processor in order to provide the needed instructions for glitching. The instruction types that were examined were limited to logical and shift instructions. The result of the code's execution (values of the registers) was sent throught the serial interface and was compared to the expected correct result in order to understand what were the effects of EMFI.

Through consecutive scans of the chip with a variety of glitching parameters, areas on the chip that were susceptible to EMFI were identified. These areas produced 3 main types of glitches. Preferred glitches, where the result was similar to the expected result but with slight differences, General glitches, where the result was unexpected and had little resemblance to the correct result, and Mutes, where the target chip was becoming unresponsive due to the EMFI and a reset was needed. The focus was on Preferred glitches because they could provide insight on the effects of EMFI. An exception to this were the exceptions thrown by the processor that were part of the General glitches.

Glitches were found to take place in the fetch, decode, execute and write-back phases of the pipeline. The results of those glitches were instruction skipping, MMU exceptions followed by a reset issued by the processor, and wrong value on the output register. The latter presented a tendency to transition bits from '1' to '0'.

Contrary to results for voltage glitching [4], no corruption in unrelated registers was found present in the Preferred glitches.

6 Future work

One could extend the work done in this research in order to observe the behavior of the target while trying to glitch arithmetic, memory and/or flow instructions. The latter could provide an interesting comparison between EMFI and voltage glitching because glitched jump instructions were yielding smaller, mostly forward jumps when voltage glitching was used [4].

Another interesting comparison would be between the whole area heat map presented in Figure 3.1, generated when trying to glitch logical and shift instructions, and a heat map that would be generated when trying to glitch memory instructions. The comparison may show that different regions could be targeted in case a memory instruction glitch is required.

Acknowledgments

The author would like to thank Riscure for providing the tools and the space to make this project a reality. Special thanks go to my supervisors Niek Timmers and Albert Spruyt for their advice and support during this research project.

References

[1] SCHMIDT, Jörn-Marc; HUTTER, Michael. Optical and em fault-attacks on crt-based rsa: Concrete results. In: *Proceedings of the Austrochip*. 2007. p. 61-67.

[2] BAYON, Pierre, et al. Contactless electromagnetic active attack on ring oscillator based true random number generator. In: *Constructive Side-Channel Analysis and Secure Design*. Springer Berlin Heidelberg, 2012. p. 151-166.

[3] SCHLÖSSER, Alexander, et al. Simple photonic emission analysis of AES. In: *Cryptographic Hardware and Embedded Systems–CHES 2012*. Springer Berlin Heidelberg, 2012. p. 41-57.

[4] SPRUYT, Albert. Building fault models for microcontrollers. 2012. URL: `http://delaat.net/rp/2011-2012/p61/report.pdf`.

[5] Riscure. EM-FI Transient Probe datasheet. URL: `https://www.riscure.com/documents/datasheet_em-fi_transient_probe.pdf`. [Accessed 30 January 2014]

[6] CARLIER, Sebastian. Electro Magnetic Fault Injection. 2012. URL: `http://delaat.net/rp/2011-2012/p19/report.pdf`.

[7] ARM. Cortex-A9 Processor. URL: `http://www.arm.com/products/processors/cortex-a/cortex-a9.php`. [Accessed 6 January 2014]

[8] ARM. The ARM Cotrex-A9 Processors. URL: `http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf`. [Accessed 6 January 2014]

[9] Riscure. EM Probe Station datasheet. URL: `https://www.riscure.com/documents/datasheet_emprobestation.pdf` [Accessed 30 January 2014]