Exploring Effects of Electromagnetic Fault Injection on a 32-bit High Speed Embedded Device Microprocessor

Master Thesis
EIT ICT Labs Master School
University of Twente

Tim Hummel

July 27, 2014

Abstract

Researchers already presented electromagnetic fault injection as an attack technique to change data and instruction execution in an electronic device. For a successful and reliable attack an attacker has to find a usable injections configuration. Therefore an attacker has to explore a range of available configuration parameters. To the best of our knowledge no summary and evaluation of glitch effect exploration techniques has been published yet. Furthermore no work has performed electromagnetic fault injection on a 32-bit target running with clock speeds above 500 Mhz. The aim of this thesis is to list and test these glitch effect exploration techniques on a 32-bit high speed embedded device microprocessor.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In recent years considerable research and development effort was invested into cryptography and securing devices. Knowledge about secure coding, cryptography and its proper implementation is becoming more and more widespread. Developers become more aware about security and privacy challenges. However even securely coded programs have to rely to a certain degree on the underlying hardware executing correctly. The bellcore [Bon01] attack explains how a single random hardware fault can expose the Ron Rivest, Adi Shamir and Leonard Adleman public-key cryptosystem (RSA) secret key in an RSA implementation based on the Chinese-Remainder-Theorem.

Fault Injection (FI) is the process of deliberately introducing faults into a device. A system can show a different behavior when a fault occurs. Traditionally fault injection is used to test the dependability of circuits in computer systems [Hsu97]. Reliability testers can use manually injected faults to simulate faulty hardware or bad environmental conditions to test the reliability of their systems. For example devices in space should be resistant to higher levels of radiation. Security researchers realized that fault injection can position vulnerable computer systems in a faulty state which can e.g. leak information or bypass security [Mau12]. Many different techniques have been proposed and successfully applied in practice, including laser FI [Wou11], mircroprobing [Sko05], temperature FI [Gov03], clock FI [Ami06] and voltage FI [Bar09]. It is known that these techniques can inject various types of faults into devices, including skipping of instructions [Sch08], bit sets, clears and toggles, word sets, clears, toggles and randomizes [Ver11]. For many years hardware based attacks received little attention and thus were a viable attack option. High costs and the longevity of some products are the reason why even after ten years of improvement in hardware security many still vulnerable products are in use today. Nowadays fault resistance is part of the standard banking security certifications for smartcards and countermeasures are implemented in such secure devices [EMV].

Electromagnetic Fault Injection (EMFI) is a FI technique with several significant advantages and

disadvantages. EMFI is based on electromagnetic induction. Positioning a closed conducting loop e.g. a metal coil, within an intensity changing magnetic field induces currents into it. A normal transistor in a processor consists of closed circuit loops. Inducing currents into processor transistors can cause all kinds logic failures and unexpected results. For a more detailed explanation of EMFI see [Aar13, p. 17]. EMFI does not require decapsulation like optical FI. Compared to optical FI, temperature FI, voltage FI and clock FI, dedicated countermeasures against EMFI are less common. EMFI is hard to use precisely and repeatedly. The EMFI injection area in our setup contains hundreds of transistors, leading to faulty unpredictable behavior, which is only determinable by experimentation. On the other hand, laser FI could be done with sufficient precision to only affect one transistor [FC14]. Another disadvantage is that, even with the same configuration of the test setup, different behavior can be observed with certain sometimes low probabilities [Mor13].

Clearly EMFI is able to introduce faults into processors, as proven by various papers and our own experiments. Literature tested EMFI for example against an Field Programmable Gate Array (FPGA) running Advanced Encryption Standard (AES) [Deh12a]. Researchers in [Deh12b] used an 8-bit microcontroller without dedicated countermeasures running AES. They managed to introduce faults and were able to characterize them and use them for a successful attack. [Aar13] used smartcards as target.

Literature studying the faults of a 32-bit microprocessor running with clock speeds above 500 Mhz, comparable to the ones found in modern embedded devices, is limited. [Mor13] and [Deh13] presented the first results and explanations of effects on 32-bit processors with slower clock speed. Velegalati et al. [RV13] injected into a "ARM Cortex-A9 Quad Core" running on 200 Mhz. We assume the higher the clock speed of a target becomes, the harder it is to accurately produce faults with a limited precision FI setup. Special techniques are required to determine in which clock cycle glitch effects occurred. Research is far from describing all possible EMFI effects in microprocessors in general and lacks experiments with higher clock frequencies.

1.2 Research Question

We wanted to study the effects introducible into modern 32-bit microprocessors by EMFI. To make our results highly comparable to standard embedded devices, we want to make our test case with a state-of-the-art embedded device processor with a standard operation speed. EMFI is cheap compared to laser FI and targets often do not implement dedicated countermeasures against EMFI, which is why this FI technique is applicable widely. This is why we want to focus our efforts on EMFI. Because the individual results can differ between different targets, we want to present our procedure, rather than the results for a specific target.

The main research question of this thesis is:

How to explore the effects EMFI causes in a 32-bit high speed embedded device microprocessor?
This question can be answered by splitting it into three sub-questions, namely:

1. What techniques are available to observe the instruction execution in a state-of-the-art

embedded device processors?

2. Which target programs are suitable for determining glitch effects?

3. Are these techniques useful for an attacker for finding potentially usable glitches in an unexplored 32-bit high speed embedded device microprocessor attack target?

There exists a variety of methods and setups used in different literature for assessing effects of EMFI. We want to list them and add own suggestions to answer sub-question 1. Although we use EMFI for our experiments, the introduced techniques are usable for a variety of FI methods. Thereby we provide researchers and security analysts with new tools for their work.

Before testing some of the identified techniques, we have to develop test programs. Sub-question 2 aims to develop these test programs, for which we would expect to see expressive results.

Finally sub-question 3 aims to verifying the usefulness of the tested techniques and the proposed test programs. A generic goal of EMFI could be to find glitches usable in an attack. If our methods are useful for finding glitches usable in an attack and with that faults in an unexplored target, we can claim that our methods are valid.

1.3 Thesis Overview

This chapter gives an introduction to the topic and presents our research questions. The rest of the thesis is structured as follows:

Chapter 2 introduces FI techniques and the normal process of using them.

Chapter 3 introduces the data acquisition sources usable to asses effects of FI. It lists the ones used in literature and proposes additions.

Chapter 4 introduces the target and target program for our experiments and gives the reasoning for our selection.

Chapter 5 introduces the hardware and software measurement setup.

Chapter 6 contains a description and first exploration of the injection parameters for the remaining experiments.

Chapter 8 presents the results obtained by using exception handlers as primary technique to asses effects of EMFI.

Chapter 7 presents the results obtained by using tracing as primary technique to asses effects of EMFI.

Finally in chapter 9 we summarize our findings and give suggestions for future work.

Appendix A contains photos of our decapsulated target processor.

CHAPTER 2

Introduction to Fault Injection

Fault injection is a group of techniques to change electronic device behavior or data. This chapter gives an overview of common fault injection techniques and how they can be practically applied.

2.1 Overview over Fault Injection Techniques

This section lists techniques, which have been successfully used to change the stored/processed data or instruction execution within a device:

Micro-probing The process of micro-probing involves placing tiny needles (called probes) on an internal signal line after decapsulation of the chip. These probes can be used to measure or overwrite a signal permanently or at a precise time during execution. [Sko05]

Temperature fault injection Heating a circuit changes its behavior, e.g. [Gov03] observed that heating a DRAM chip up to 100°C caused several flipped bit errors in the memory.

Voltage fault injection The supply voltage of a device is lowered or raised in short pulses (so called glitching) or permanently (so called underfeeding/overfeeding). This can introduce several behavior changes, as longer propagation times on bus lines and flip flops, might fail to hold their values. With the voltage underfeeding, logic levels are not able to raise to their correct level in the specified time and might get interpreted wrongly. [Bar09]

Clock fault injection A similar technique to voltage FI, but on the clock line instead of the power supply line. Clock FI can lead to different calculation outcomes and to incorrect data writes. [Ami06]

Laser fault injection Transistors are inherently vulnerable to manipulation by photon injections. A strong laser beam can e.g. open or close a single transistor. In comparison to a standard light source, a laser can be applied very precisely, which makes it possible

to target single transistors [Wou11]. The major drawback is that the chip has to be decapsulated and sometimes modified for the laser to reach the transistors.

Ion radiation fault injection Radiation as fault injector shows similar behavior as laser fault injection, including the need for decapsulation. However it is much more imprecise and requires handling radiation. [Kar95]

UV light fault injection Some one-time-programmable memory cells are erasable by UV light including some security fuse registers. This technique has been used to purposely erase one-time-programmable memory in former microcontroller generations and making it programmable again. [Sko05]

Focused ion beam The focused ion beam technology allows removing and adding metal with an highly focused beam of ions [Orl03]. The technology was primarily used for research and debugging in the semiconductor industry and is now also used for security research. The technique offers plenty of alteration possibilities like connecting two signals, cutting a wire or adding pads for micro-probing. This technique has enough precision to change single wires, even in modern chips with their high density.

Electromagnetic fault injection Positioning a closed conducting loop e.g. a metal coil, within an intensity changing magnetic field induces currents into it. A normal transistor in a processor consists of closed circuit loops. Inducing currents into processor transistors can cause all kinds logic failures and unexpected results. For a more detailed explanation of EMFI see [Aar13, p. 17].

2.2 Practical Fault Injection Usage

```
1  boolean checkPin(char* pin) {
2    charArray correct_pin = {1,2,3,4}
3    for (i=0; i<length(correct_pin); i++)  {
4      if (pin[i] != correct_pin[]){
5        reducePinTryCounter()
6        return false
7      }
8    }
9    return true
10 }
```

Listing 2.1: Pseudocode of a seemingly secure password check function, but only as long as the underlying hardware executes correctly

Listing 2.1 shows a seemingly secure pseudocode function for checking if a pin is correct. The function compares all the pin bytes. If the pin byte and the correct pin byte do not match, the function reports an incorrect pin and additionally decreases the pin-try-counter.

However, this function is only secure if the underlying hardware executes correctly. FI can alter the data and instruction execution.

This example has multiple potential points of failure, including but not limited to: The counter variable could be increased so not all pin bytes are checked. The reducePinTryCounter function could be repeatedly skipped, allowing an attacker to try all possible combinations for the pin. An instruction opcode could be altered while loading it from memory, translating it into something completely different. The pointer to the correct_pin could be changed to also point to the pin, so that the check would compare the pin to itself. Processor register content could be manipulated in single bits (e.g. set them, clear them or toggle them) or entirely (e.g. set a whole register, clear a whole register or fill it with arbitrary or seemingly arbitrary data) and thereby change progams' variables and pointers.

Additionally faults can have different timely behavior:

Permanent faults are irreversible damages to a component. The fault can only be removed by removing or replacing the faulty component e.g. burning away a part of a circuit permanently damages it.

Temporary faults are recoverable and only change a circuit part behavior e.g. a single transistor conductivity for a limited time e.g. the next reboot.

Transient faults are recoverable and only change a circuit part behavior e.g. a single transistor conductivity during the injection time. However the fault may propagate and leave the system in an erroneous state.

Which precise behavior changes like instruction skips, variable changes, etc. are achievable, depends on the low level structure of a given target. The achievable faults and how long they last is usually determined by experimentation. To perform a successful and usable FI, an attacker has to figure out the exact conditions. Each fault injection technique has several configurable parameters. A clock glitch injected via a pulse in the clock line of a smartcard e.g. can have different pulse shapes, a different timing, a different absolute intensity etc. Finding all the parameters needed for a desired fault often requires reducing the parameter range with educated guesses and then iterating through the remaining parameter space until a usable FI is observed. Searching usable faults in our available parameter space is one of the main task we perform in this thesis. Special test programs discussed in section 4.2 and techniques in chapter 3 can make successful glitches more visible and likely, thereby accelerating our search. The parameters found with a test program can then be used to attack the checkPin function displayed in listing 2.1.

CHAPTER 3

Overview of Data Acquisition Sources

This chapter introduces the sources of information, which can be used to observe the effects FI causes within a processor. This chapter lists the techniques used in related literature and proposes additions.

The general goal of every technique is to help understand what changes between a normal execution of a program and an execution influenced by FI. In a perfect observation setup the complete processor could be monitored constantly including all wires and all transistors. Unfortunately complete observability is not possible in any common processor. Several researchers [Aar13; Deh12b; Mor13; Spr13] try to observe a target's program execution by comparing the result of a normal and a glitched calculation or by comparing processor state information after a normal and a glitched program execution. The relevant state in an Advanced RISC Machines (ARM) processor could e.g. be the processor's content of register R0 to R15, CPSR and selected regions of the memory.

Our scope is limited to techniques available for the ARM architecture, because it is a dominant architecture for embedded devices and looking into more architectures would have been outside the timely scope for this thesis. The ARM architecture is a well-documented and widely used platform for embedded devices. Similar techniques as the ones described in this chapter might exist in different architectures e.g. MIPS, but comparing them would have exceeded our time constraints.

3.1 Software

Most literature instrumentalizes the software running on the target directly [see Aar13; Deh12b; Spr13]. The software can execute a simple calculation and the results get transferred to a host machine. If the result changes during FI for a normally deterministic execution, we assume that a glitch has occurred. For example adding up the numbers from 1 to 10 should always result in 55. If we perform FI during this calculation and the results instead is 54, we can be sure that the FI has successfully produced a fault.

Additionally to producing a result, software can directly collect state information by reading registers and memory within the target. The collected information can be transferred to the measurement host e.g over Universal Asynchronous Receiver Transmitter (UART) [Spr13]. This can be used to observe register corruption [Spr13] or other faults. Transferring all registers, not only the ones with the calculation results, lets us also observe some faults not necessarily manifesting in the result of a calculation. For example if a calculation only uses R0 and a random bitset occurs in another register, one could observe it only by collecting the state of all remaining registers.

Relying only on the software for providing state information has limitations. The data collection and transfer process changes the state of the processor and potentially destroys state information. Additionally the state information can only reach the host, if the code responsible for transferring the result can be executed correctly. If the processor ends up in an unrecoverable state due to a glitch (e.g. an arbitrary jump into another memory region), it cannot answer anymore. A solution for this problem is using exception handlers for some faults, see section 3.2.

3.2 Exceptions

The normal program flow of sequential instructions, branches and subroutines within a target can be diverted by external or internal events. Events like interrupts or non-executable instructions cause the processor to halt the current execution and execute an exception handler. Exception handlers are predefined subroutines executed in such events. Their start addresses are listed in the exception vector table. If an event is encountered the processor preserves the current state and executes the subroutine at the address listed in the exception vector table. The exception handler and exception vector table can both be defined by the developer.

Table 3.1 lists the common exceptions events in an ARM microcontrollers [Exc]. Monitoring the occurring exceptions gives us information about the processor behavior during FI [Mor13]. For example encountering an "Undefined Instruction" exception might mean that the glitch changed an instruction to a non-executable one. On each exception entry the last executed Program Counter (PC) value added by 4 gets stored in the Link Register (LR). Depending on the exception and due to pipelining this must not be the exact instruction causing the exception, but a close one [A8t]. Moro et al. used the LR value to determine the instruction they glitched in their experiments[Mor13].

Table 3.1: The common ARM exceptions with occurrence reason. Source: [Exc]

Exception	Occurrence
Reset	Occurs when the CPU reset pin is asserted. This exception is only expected to occur for signaling power-up, or for resetting as if the CPU has just powered up. It can therefore be useful for producing soft resets.
Undefined Instruction	Occurs if neither the CPU nor any attached coprocessor recognizes the currently executing instruction.
Software Interrupt (SWI)	This is a user-defined synchronous interrupt instruction, which allows a program running in User mode to request privileged operations which need to be run in Supervisor mode.
Prefetch Abort	Occurs when the CPU attempts to execute an instruction which has prefetched from an illegal address, i.e. an address that the memory management subsystem has determined as inaccessible to the CPU in its current mode.
Data Abort	Occurs when a data transfer instruction attempts to load or store data at an illegal address.
IRQ	Occurs when the CPU's external interrupt request pin is asserted (LOW) and the I bit in the CPSR is clear.
FIQ	Occurs when the CPU's external fast interrupt request pin is asserted (LOW) and the F bit in the CPSR is clear.

In some ARM cores are extra registers to provide higher verbosity in case of an exception. In the ARM Cortex A8 the prefetch abort exceptions sets the CP15_INSTRUCTION_FAULT_STATUS and CP15_INSTRUCTION_FAULT_ADDRESS register and the data abort exceptions sets the CP15_DATA_FAULT_STATUS and CP15_DATA_FAULT_ADDRESS register. Both FAULT_STATUS registers [A8t, 3-63 ff.] contain codes defining the precise reason for the most recent exception. Precise reasons are among others, alignment fault and permissions faults for data abort exceptions or permission fault and external abort for prefetch abort exceptions. The corresponding FAULT_ADDRESS registers preserved the instruction address which caused the last exception. The FAULT_ADDRESS registers can be used to tell which instruction was affected by a glitch, if it created an exception.

The exception handlers can be programmed to send state information including R0-R15, LR, FAULT_STATUS and FAULT_ADDRESS registers to a host machine and thereby a software-only program can still return data for some glitches that divert the execution flow.

3.3 OCD

On Chip Debugger (OCD) can be used to debug programs running directly on the processor without an underlying Operation System (OS). Standard features of OCD are reading and writing memory and registers, halting, continuing, resetting using breakpoints and watchpoints and loading program images [Ope]. OCD is a common feature in ARM processors. OCD usage requires additional hardware and software. The OCD is usually connected to the Joint Test Action Group (JTAG) port, therefore a so-called JTAG emulator is required. OCD can be used

to retrieve processor state information. Even if the software on the target is not able to respond anymore due to a glitch, the OCD might still get data. For using OCD to transfer data to a host, opposed to using the target software alone, one does not need to alter the processor registers. Moro et al. used OCD to build their FI setup [Mor13].

3.4 Tracing

Tracing is a form of logging a program execution at low level. Listing 3.1 shows an example program's source code. In an optimal trace setup we could observe for each processor cycle, which instruction was executed at which memory location and whether it was successful.

```
403040a0: mov R4, #0x20000
403040a4: movw R5, #0xC194
403040a8: movt R5, #0x4804  <---tracing configured to start here
403040ac: str R4, [R5]
403040b0: add R0,R0,#1
403040b4: add R1,R1,#11   <---tracing configured to end here
403040b8: add R0,R0,#1
403040bc: add R1,R1,#1
```

Listing 3.1: The assembly code for the trace given in listing 3.2. The trace was configured to only trace instruction including 0x403040a8 to 0x403040b4

To provide this functionality the ARM architecture specifies optional trace macrocells for real time trace acquisition during runtime. Macrocells are optional and only present in an ARM chip if they have been implemented by the chip designer. According to [Gmb, p. 49] the most common trace macrocell is the Embedded Trace Macrocell (ETM)v3. ETMv3 should provide the ability for "full instruction traces" [Etm], meaning the whole program flow should be observable including all jumps, conditional executions etc. The trace data is stored in a special format, which can be decoded and interpreted by knowing the program code in advance.

Unfortunately during FI instructions might be glitched to behave like other instructions. Therefore, although the program code is known in advance, it might not be the one executed by the CPU. This is equivalent to not knowing the program code in advance. Thereby all common Integrated Development Environment (IDE)s following the ETM specification will produce faulty data. Alexander Shishkin developed etm2human [Shi] an opensource program to parse the raw trace data to as far as possible without knowledge of the program code. We contributed to his program by extending it to decode cycle accurate traces (tracing, which logs every processor cycle) for the Texas Instruments Sitara AM3358AZCZ100. To illustrate which data is obtainable with this limited tracing functionality, listing 3.2 shows the parsed trace data produced by etm2human for the assembly routine in listing 3.1.

```
trace flow started at 403040a8, cycle 0
insn at 403040a8: X cycle: 0 cond: PASS
insn at 403040ac: X cycle: 172 cond: PASS data_addr: 4804c194
insn at 403040b0: X cycle: 172 cond: PASS
insn at 403040b4: X cycle: 173 cond: PASS
```

Listing 3.2: A short example trace containing 4 instructions decoded with etm2human.

The trace start form the current PC value and a cycle counter initialized to zero. The remaining trace data only contains cycle offsets and whether an instruction passed its condition or not. The instruction opcode is not contained in the trace and usually has to be derived from the source code. The displayed instruction memory address is only relative and sequential to the start of the trace. It is wrong as soon as the processor encounters a branch instruction. From the sourcecode we could know that a successfully executed instruction at a memory location which contains a branch means that the target now executes code from a different location, but from the raw trace data we do not. Overall the information from tracing is quite limited but might be sufficient for some experiments.

Additional optional features are data address and data value tracing/observability for Load Store Multiple (LSM) instructions. In the target chosen in chapter 4 only data address tracing is implemented.

ARM specifies also a AHB Trace Macrocell (HTM) for tracing the Advanced High Performance Bus (AHB) bus. Among others the AHB connects the flash to the main core. Monitoring the AHB would be the logical next step to verify AHB bus transfer glitches introduced in [Mor13], however we were not able to find any device implementing the HTM.

Neither of the trace macrocells seems to be used for FI research so far.

3.5 External Clock

Moro et al. observes the processor clock while injecting faults into a processor. They configure the processor to expose the internal clock signal on an Input/Output (I/O) pin [Mor13]. Monitoring the processor clock enables them to check in which exact moment within a single clock cycle a glitch is injected and enables counting into which clock cycle relative to some trigger event a glitch is injected. Several ARM microprocessors offer similar options. Unfortunately for the processors we checked (Texas Instruments Sitara AM3358AZCZ100 and Texas Instruments OMAP3530) it is not possible to expose the processor clock signal directly. There exist speed limitations for the I/O pin, the clock signal has to pass through some clock dividers first, additionally the I/O and core clock are generated by two different Phase-Locked Loops (PLL).

3.6 Additional Options

The Cortex A8 microprocessor (A8) contains additional registers to supply debug information. For example resettable counters for counting elapsed processor cycles. The Cortex M3 contains additional counters for folded instructions, elapsed sleep cycles, exception cycles and LSM cycles [p. 8-3 Cou].

These registers might be used as supplementary information or as consistency check for some experiments.

3.7 Reliability

It is unclear how the information sources themselves are influenced by FI. The ETM and OCD need parts of the internal logic operational to function properly. It is unclear under which conditions the data produced by these components is reliable and usable for glitch analysis. Chapter 7.2 tries to test the reliability of tracing through experimentation.

3.8 Selection

For our own experiments we decided to use test programs with exceptions handlers. We have an extended experiment setup that also uses tracing and OCD usable with the same test programs with exception handlers. Tracing were to the best of our knowledge never used in literature. Exceptions were not used to the extent of our experiments in literature before. We did not focus on OCD unless needed for tracing, because we have to avoid the efforts of analyzing these additional data due to our timely scope. External clock and cycle counter usage was not used, because this would also have broken our timely scope.

CHAPTER 4

Target Selection

This chapter introduces and gives the reasoning for the choice of target device and the programs running on the target device.

4.1 Target Hardware

The target has to fulfill basic criteria, mainly it has to be possible to inject faults into it, it has to be an ARM and it has to implements trace functionality. We selected the Beagle Bone Black (BBB) development board. The BBB has an AM3358 family processor, the Texas Instruments Sitara AM3358AZCZ100 [Ins] microprocessor. It contains an A8 running with up to 1 Ghz clock speed. This 1 Ghz maximum clock speed was used in all our experiments. Figure 4.1 shows a top view of the board, the processor is in the square package "U5" in the middle of the board. This target fulfills all necessary requirements needed for glitchability and glitch effect analysis.

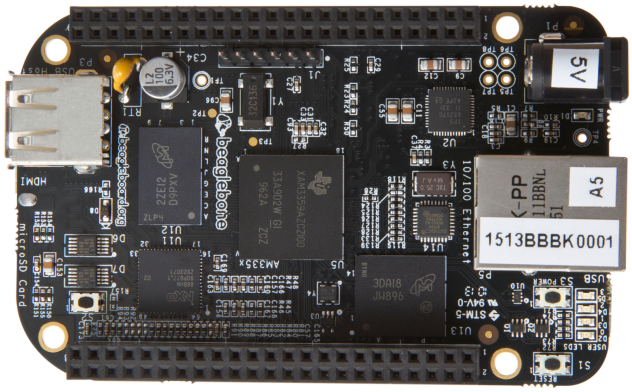

Figure 4.1: The BBB is the development board used in this thesis. The processor contains an A8. The processor is in the square package in the middle of the board.

In the previous chapter, we identified the most promising techniques for collecting glitch information as tracing and exception usage. Exception Handling and OCD(which is required for retrieving the trace data) is a standard feature in every regular ARM core. Our hardest to fulfill criterion was the tracing functionality. Instruction tracing is an optional feature available only if an ETM or a similar product is implemented in the processor. Software is required to configure the target for tracing and read out and parse the trace data. Depending on the targets features, the data is either stored in a dedicated memory Embedded Trace Buffer (ETB) or readable on an external trace bus. Using the external trace bus requires additional hardware, which is expensive, also it is unknown how it performs in combination with FI. The ETB is inexpensive to use, because it can be accessed by OCD. By using an ETB, we can also be sure to get the raw trace data directly and not the potentially augmented data from proprietary trace bus readers. Implementing an own program from scratch to set the trace configuration and dumping the ETB would be out of the scope of this thesis. Therefore we are bound to the only publicly available implementation we found, Code Composer Studio (CCS) from Texas Instruments (TI). CCS features an Application Programming Interface (API), which allows to integrate tracing into our test setup. Additionally to use and modify low level features like tracing, detailed documentation of the chip is required. This limits us to processors which are thoroughly documented, compatible with CCS and have an ETB.

The ETM exists with different feature sets. The most relevant difference is, whether or not data address and/or data value tracing is supported for load, store and other LSM instructions. For each LSM instruction we could see the target/source memory location and/or the data value. At least having one of those could give interesting insights in LSM instructions. [Gmb] provides a table showing which features might be implemented in which ARM core. The value of tracing is higher in cores with data value and address tracing. This excludes the Cortex M processors. Our analysis shows that the features of the ETM are often not documented in the official processor datasheets, but can be extracted by reading the internal ETM feature registers manually. The BBB has an ETB and an ETM providing data address tracing, but no data value tracing.

For glitching the target the core package should be directly visible. Package on Package (PoP) packages are a form of reducing the distance between memory and core by stacking the memory package on top of the core package. Riscure's experience shows that these are significantly harder to glitch.

The BBB fulfills all our basic criteria. There might be better targets, for example targets additionally implementing data value tracing, but obtaining ETM features is time consuming and expensive. Also for the BBB we had to manually readout the feature core's feature registers to get this information. Therefore we decided to just use the first target matching the basic requirements, instead of spending more time on finding a one with e.g. additionally data value tracing.

We analyzed different targets, which did not fulfill our basic criteria. Table 4.1 gives an overview over the different potential targets we investigated.

Table 4.1: The potential targets or target groups we considered for this thesis and whether they fulfill all basic criteria.

Target name	Result
32L152CDISCOVERY used in [Mor13]	no ETM
STM32F103ZG	not compatible with CCS
Cortex-M processors	no data tracing [Gmb]
BeagleBoard	PoP package
BBB	fulfills all criteria

4.2 Target Program

We developed seven different target programs. Each one was developed to test the effects on a single instruction type or a variety of instruction types only, as opposed the huge variety which exist in ordinary programs. We suppose by only testing tiny parts, we can derive conclusion much easier and can still see the whole pictures by combining the results . By target program we mean only the instructions we want to glitch, not the entire actual program, which also includes the wrapper. The wrapper contains other instructions needed, for example for communication, exception handling and configuration of the target.

Each target program executes a calculation using the registers R0 and R1. The wrapper initializes all registers (CPSR, R0-R12) to a known value before each calculation. R0 to R4 and R6 to R12 are initialized to 0xff00ffff, 0xff01ffff ... 0xff12ffff, if not stated otherwise. R5 is initialized to 0x4804C194, an I/O register address, for setting a trigger signal high. The trigger signal is a signal needed for our measurement setup to trigger the injection. The wrapper also transfers all final values including the values of R0 and R1 to the host computer via UART after each calculation. The wrapper also contains exceptions handlers, which immediately transfer exception type, all register values and the exception register values to the host computer, if an exception occurs. Additionally the wrapper sets an I/O pin needed for triggering to high before each calculation and to low after each calculation. A calculation can be initiated, by sending an UART command to the wrapper.

The following target programs were developed to run within the wrapper:

everythingloop The everythingloop target program contains instructions from several categories: LSM, arithmetic and branches (mov was unintentionally omitted). We assume that glitches in different instruction types manifest in the processor state in different forms. For the initial experiments we wanted a program which is easily glitchable and in which glitches could manifest easily in the result values. This program can be used to test if a target is glitchable at all and under which parameters like location and power it works best. The target program in listing 4.1 consists out of a loop with 0x5000 iterations, which loads, increments and stores back a value at someMemoryLocation. A change in any of these instructions or the data values used most likely manifests in the output. The expected result values are be R0 = 0x5001 and R1 = 0x5001.

```
;initialization
mov R1, #0
movw R2, someMemoryLocation
movt R2, someMemoryLocation
str R1, [R2]
mov R0, #0
;R3, R4 and R6 to R12 are initialized to 0

;target program
loop
  ;increment data value
  ldr R1, [R2]
  add R1, R1, #1
  str R1, [R2]
  ;loop management
  add R0, R0, #1
  cmp R0, #0x5000
  bls loop ;lower or same
```

Listing 4.1: The everythingloop is an easily glitchable program

nopsled The nopsled target program in listing 4.2 consists of 20 nop instructions. However a real nop instruction does not exist in the ARM instruction set. ARM itself uses "mov R0, R0" for that purpose [Lim]. Initial experiments quickly revealed that this instruction is influencable. The effects on this instruction are studied with the movsled target program. So the nop has to be replaced with an instruction we assume has no effect even when glitched. We think that the conditional instruction "movne R0, R0" very likely has no effect, because both the condition has to be glitched and the actual executed instruction. R0 and R1 are initialized to 0xffffffff. Because the nop instructions should not change the state of the registers, this target program might reveal faults unrelated to the instructions used. The results from measurements with the nopsled compared with other instruction target programs could reveal the distinctive faults for other instructions.

```
cmp R0, R0 ;set condition flags to notEqual

;target program
20x
movne R0, R0
```

Listing 4.2: The nopsled target program.

addsled The addsled target program in listing 4.3 consists of 20 add instructions, alternately incrementing R0 and R1 by one. To determine the precise effects of a glitch we want to be able to check how glitching single instruction namely here the "add with constant" could change the processor state. The expected result values are R0=0xa and R1=0xa.

```
;initialization
mov R0, #0xffff01f0
mov R1, #0xffff10f0

;target program
10x
add R0, R0, #1
add R1, R1, #1
```

Listing 4.3: The addsled target program.

movsled With the intention of studying the effects on mov instructions, the movsled target program in listing 4.2 consists of 20 mov instructions. R0 and R1 are initialized to 0xffffffff. Because moving a register to itself should not change its state, the target program might reveal faults in the moving of the value.

```
;initialization
mov R0, #0xffff01f0
mov R1, #0xffff10f0

;target program
20x
mov R0, R0
```

Listing 4.4: The movsled target program.

branchsled With the intention of studying the effects on conditional branch instructions, the branchsled target program in listing 4.5 consists of 20 conditional branch instructions. The branches should skip writing of R0, so R0=0xFFAA should only be observed, if a branch condition has been glitched.

```
;initialization
mov R0, #0xffff01f0
mov R1, #0xffff10f0

;target program
20x
bne jumpgoal
mov R0, #0xffaa

jumpgoal
mov R1, #0xffbb
```

Listing 4.5: The branchsled target program.

comparesled With the intention of studying the effects on compare instructions, the comparesled target program in listing 4.6 consists of 20 conditional branch instructions. The branches should skip writing of R0, so R0=0xFFAA should only be observed, if a compare or a

branch condition has been glitched.

```
;initialization
mov R0, #0xffff01f0
mov R1, #0xffff10f0

;target program
20x
cmp R0, R0
bne jumpgoal
mov R0, #0xffaa

jumpgoal:
mov R1, #0xffbb
```

Listing 4.6: The comparesled target program.

storesled The storesled target program can be used to study effects on store instructions in combination with an initialization of the registers needed for the store instruction. It is actually the wrapper program not filled with a target program, because the wrapper immediately after executing a target program starts storing the registers for sending them to the host.

```
;target program
;none

;trigger off
mov R12, #0x20000
movw R5, #0xc190
movt R5, #0x4804
str R12, [R5]

;storeOutput
movw R12, memoryForR0
movt R12, memoryForR0
str R0, [R12]
movw R12, memoryForR1
movt R12, memoryForR1
str R1, [R12]
movw R12, memoryForR2
movt R12, memoryForR2
str R2, [R12]
;and so on for the remaining registers except R5
...
```

Listing 4.7: The storesled target program.

4.3 Summary

We selected a target and presented a series of example programs that we consider highly suitable for analyzing glitch effects. Different test programs are used to study effects on different instructions. Together the individual test programs cover as much of the common instruction types as time allowed.

CHAPTER 5

Measurement Setup

This chapter describes our measurement setup. The setup consists of a hardware part and a software part.

5.1 Hardware Setup

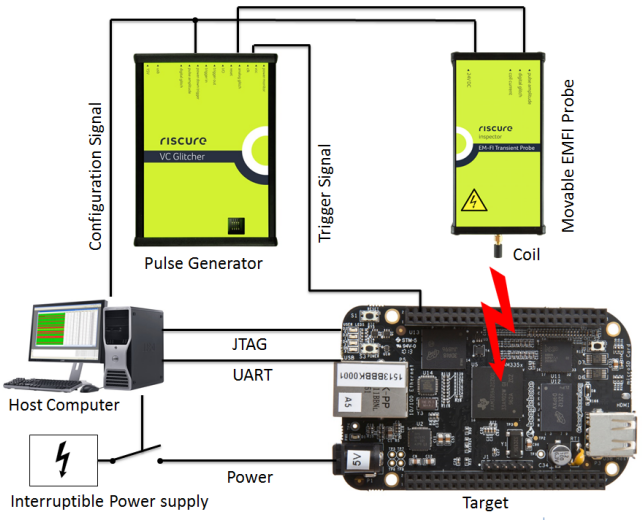

Figure 5.1: Functional schematic of the measurement setup

The EMFI setup shown in figure 5.1 and the figures 5.2 and 5.3 consists of the BBB with the target processor, a movable EMFI probe, a pulse generator, an interruptible power supply and a host computer. The target program with the wrapper is running on the target processor. The target processor is positioned below the movable EMFI probe. The tip of the probe is a single-loop metal coil with a diameter of 1.5 mm. The EMFI probe is connected to the pulse generator. The EMFI probe discharges a capacitor bank into the coil as soon as it receives

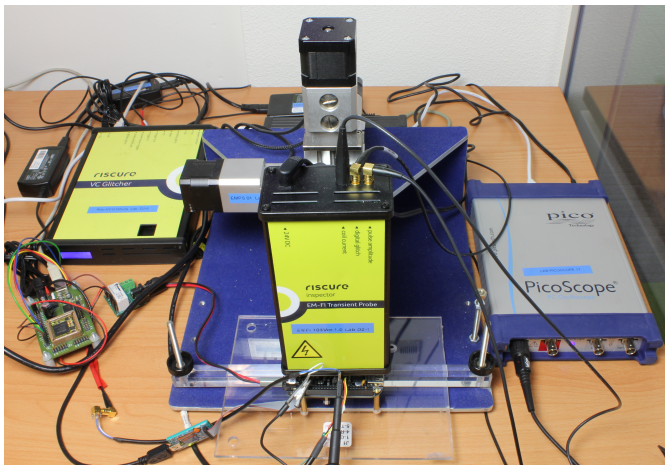

Figure 5.2: Overview photo of the measurement setup. The EMFI probe is fixed to a XYZ stage in the middle of the photo. The pulse generator and the interruptible power supply are left of the XYZ stage. The oscilloscope for measuring e.g. the trigger signal is positioned on the right.

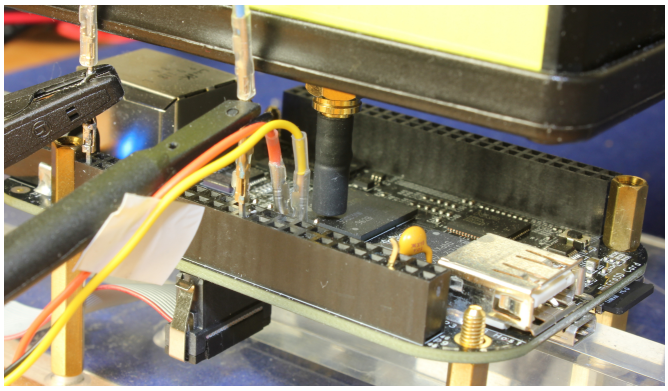

Figure 5.3: Close-up photo of the injection coil. The coil is positioned as close as possible over the processor package, without touching it. The whole EMFI probe including the coil can be moved in all 3 dimensions by the measurement host computer. The visible wires are the trigger signal, the UART wires and the measurement probe of the oscilloscope.

a pulse from the pulse generator. The capacitor bank discharge into the coil and creates an Electromagnetic (EM) pulse. The pulse generator waits a definable time and emits a pulse as soon as a trigger signal from the target is detected. This delay, between receiving the trigger signal from the target and the pulse generator emitting the pulse, is the configuration parameter called Glitch-Offset. To avoid unnecessary time delays and ensure maximum relation between the executed instructions within the target processor and the FI, the trigger signal comes directly from the target instead of passing through or being generated by the host computer. The target wrapper program is responsible for setting the trigger signal to high immediately before the target program execution begins. The target is connected to the host computer via UART for communication and via JTAG for the OCD. The target's power supply can be interrupted by the host computer to force a reboot of the target. Both the movable EMFI probe[BVa] and the

pulse generator [BVb] are products of Riscure and are configured by the host computer. The switchable power supply is a relay attached to a generic 5 V power supply controlled via UART commands by the host computer.

An oscilloscope (PicoScope 5203) is used to measure the trigger signals and glitch signal when required.

The main limitation of our setup is the temporal precision. In a perfect setup we would be able to repeatedly emit the pulse at a specific moment in time within a single processor cycle. This could be interesting, because [Deh13] observed different behavior when glitching different times within a single processor cycle. Because our target runs with 1 Ghz, every processor cycle is 1 ns long. Our measurement setup has a delay after receiving the target trigger of roughly 85±3 ns. Additionally there is an unknown delay between the instruction cycle in the target processor for setting the trigger signal to high and the trigger signal being high on the I/O pin. Section 8.1 tries to determine this precision experimentally.

5.2 Software Setup

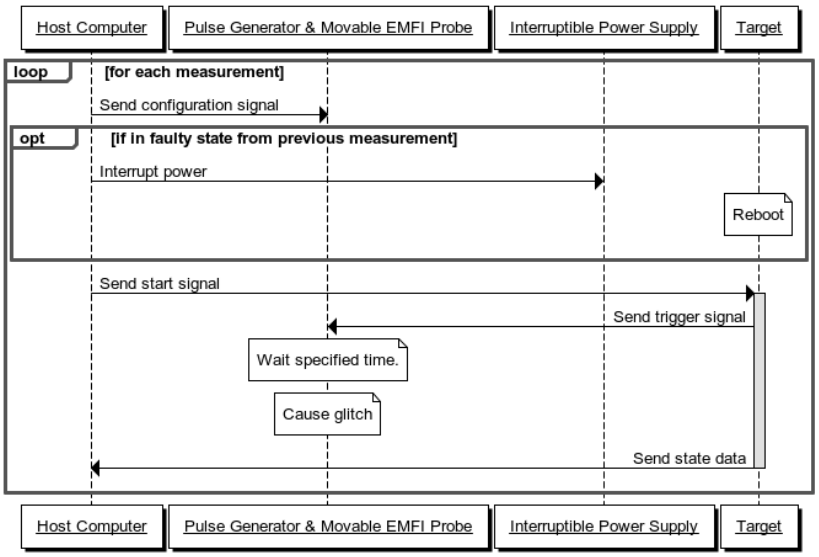

Figure 5.4: A sequence diagram illustrating the interaction, between the individual hardware components.

For each testrun with our setup, we specify a target program to glitch and a set of injection parameters. For each configurable injection parameter we can either specify a range or set a fixed value. The measurement setup then performs a testrun with those parameters autonomous. Each testrun consists of single measurements. Each measurement is a single execution of the test program in the target, after which the state information is collected and stored. For example a testrun with the addsled test program could create a database similar to table 5.1. One limitation here is that we do not have any means of checking if the communication from our wrapper program is not faulty and a register value really has a certain value. However, we set the timeframe to inject glitches into small enough to be within the expected run time of our

target program to not affect the wrapper program's communication.

Table 5.1: An example testrun results database for the addsled test program with five single measurements. Id 2 contains an abnormal value for R1, so might be a successful glitch. The glitch parameters are described in table 6.1.

ID	Probe Position	G.-Offset	G.-Intensity	State Information after calculation
0	123321, 321123	16 ns	78 %	R0 0xa R1 0xa R2 0x0 R3 ... R15 ...
1	123321, 321123	38 ns	78 %	R0 0xa R1 0xa R2 0x0 R3 ... R15 ...
2	123321, 321323	24 ns	78 %	R0 0xa R1 0xff00a R2 0x0 R3 ... R15 ...
3	123321, 321323	12 ns	78 %	R0 0xa R1 0xa R2 0x0 R3 ... R15 ...

For each measurement in a testrun, first the measurement parameters like probe position and glitch power are configured by the host computer. If necessary, the power supply is interrupted to get the target into a known state. After a reset the target boots from SD card and runs the wrapper and test program automatically.

We do not reset if the result state of the previous measurement appears unaffected, because we assume the glitch did not cause any relevant effect. Not resetting saves time between measurements, it enables us to perform considerable more measurements in the course of this thesis. After booting or a completed measurement, the target waits for the host to request a new measurement over UART. As soon as the measurement starts the target sets the trigger signal high, executes the target program, sets the trigger signal low and sends its state data to the host over UART. At the same time the pulse generator detects the trigger and fires after a configurable amount of waiting time, called Glitch-Offset. Figure 5.4 illustrates the testrun flow.

An extended measurement setup is available for measurements with tracing and OCD. We additionally use the CCS API to trace a part of the program on the target. The program is loaded with the OCD, instead of from the SD card, as required by the CCS API for tracing. After the glitch was emitted and the target program executed completely, the trace data is retrieved from the ETB via OCD. In addition to the data in table 6.1, tracing data and state data in the form of register content is collected with OCD. OCD can be used as a complementary source for register values, to confirm that the ones transferred by the wrapper script are correct.

According to [Mor13] and our own experience, EMFI measurements with the same parameters can show different behavior. Therefore to analyze EMFI, many measurements have to be made to observe as much possible behavior as possible. Therefore the measurement setup has to produce enough measurements in a decent time. The average speed for a single measurement with our setup is given in table 5.2. To increase the speed to boot from the SD card we modified an x-loader and u-boot [Eng] bootloader chain, with all debug messages and the timeout to enter the boot menu removed. This bootloader chain configures the target hardware and then immediately branches into the selected test program.

Table 5.2: Approximate average speeds for single measurements for our measurement setup and the addsled target program. Using tracing requires much more time, because the CCS API and OCD needs to be used.

Single measurement time	Without reset	Including reset
Without tracing	420 ms	2500 ms
With tracing	2000 ms	19500 ms

CHAPTER 6

Study of Fault Injection Parameters

Our EMFI setup has five parameters to configure for getting and increasing the chance for a successful glitch. In this chapter, we want to perform an initial analysis of the parameters our measurement setup offers.

Table 6.1 lists the parameters configurable in our measurement setup.

Table 6.1: The parameters configurable in our measurement setup.

Parameter	Description
XY-Position	The position of the injection probe relative to the top surface area of the target.
Z-Position	The distance of the injection probe from the top surface of the target
Glitch-Offset	The time the setup waits after receiving the trigger before emitting the pulse in ns. We do not include the delay the setup has per default, i.e. a 0 ns offset could already be a 100 ns delay. A 10 ns offset then likewise means a total delay of 110 ns.
Glitch-Intensity	Determines the intensity of the pulse. It configures the maximum voltage across the injection coil. It thereby influences the change of magnetic flux and the currents induced into the target. The value is a percentage of 450 V.

6.1 Z-Position

The Z-Position behaves like the Glitch-Intensity [Mor13] and has to be changed to increase or decrease the intensity of the glitch more than the Glitch-Intensity can, additionally it changes the area of effect. We never required a higher Glitch-Intensity, therefore the Z-Position was permanently set to the same value for all our experiments. The probe distance measured is 0.6 mm.

6.2 Glitchability and Position

In a first experiment we verify that the target is indeed glitchable with our setup. The program used is the everythingloop, because of its expected easy glitchability as explained in section 4.2. We set the Glitch-Offset to an arbitrary fixed value. Glitch-Voltage was fixed to 70 %. The X and Y-Position was changed stepwise, so that injection was performed on each position in a 100 by 100 grid on the target. On each position injection was performed 15 times. We differentiate between three types of results:

Expected Answer/Green The answer from the target does not differ from the expected result, i.e. no glitch occurred or at least not one observable from our setup.

No Answer/Red The target did not answer with a result. This means the target execution halted or ended up in an unrecoverable state.

Abnormal Answer/Yellow The target answered with an answer differing from the expected. This is the desired answer for an attacker. This will be later differentiated finer into Abnormal Answer and Exceptions.

Figure 6.1 shows that distinct areas of the targets surface are more sensitive to EMFI. Only abnormal answers (yellow) are useful, assuming the goal of an attack is to inject a fault, without permanently terminating execution. Two roundish areas are glitchable, interleaved with islands of stability. Usable glitches (yellow) occur together or within the border region of no answer glitches (red). Additionally we see some lonely glitchable positions.

Figure 6.2 repeats the same experiment on the highly glitchable areas using the whole 100 by 100 grid resolution only for this area and 18 injections per position. This follow-up experiment confirms our earlier observations, but shows that the sensitive regions are interleaved at more locations than visible in the first experiment. The edges of the sensitive areas seem to become more precise the bigger the measurement grid is. Some positions have different results when injected into multiple times, so for finding all possible faults every position has to be injected into multiple times.

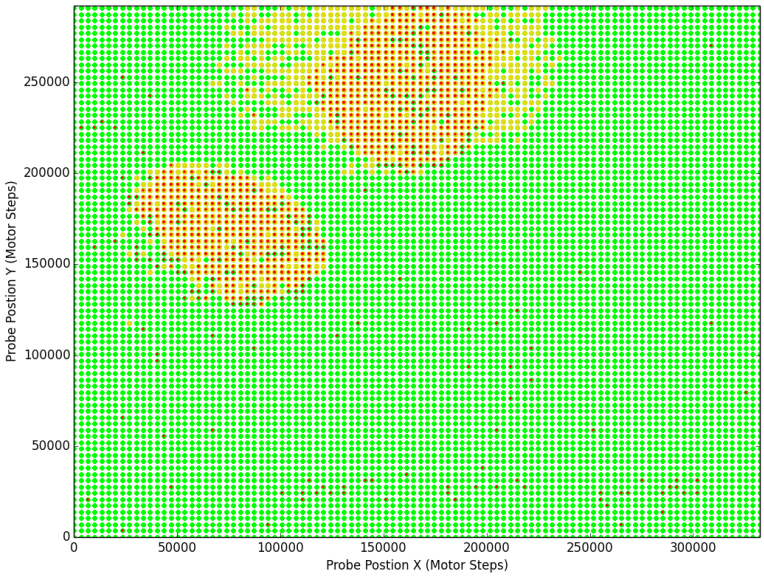

Figure 6.1: An EMFI while running everythingloop test program over the whole surface of the chip. Green dots represent expected results, red dots a not answering target and yellow dots abnormal answers. Dots on top of each other mean that several different results occurred for this location.

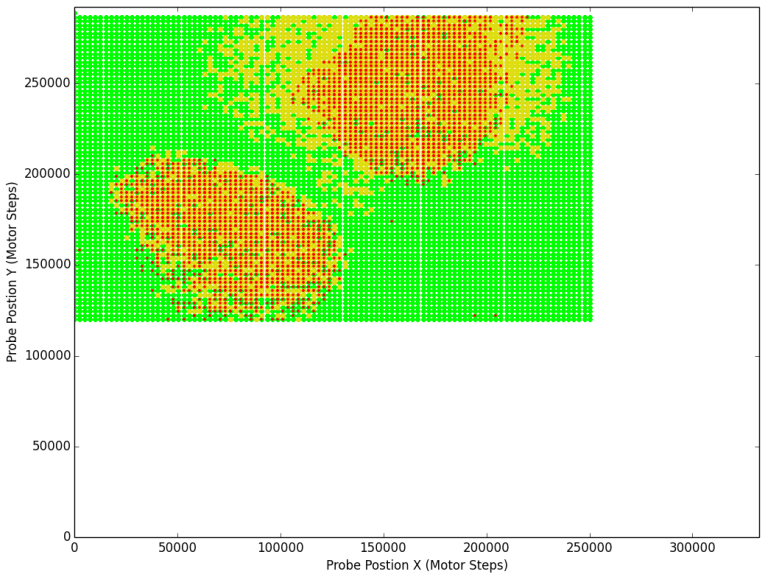

Figure 6.2: An EMFI while running everythingloop test program over the highly EMFI-sensitive area of the chip. Green dots represent expected answers results, red dots a not answering target and yellow dots abnormal answers. Dots on top of each other mean that several different results occurred for this location.

6.3 Conclusion

We were able confirm that our target is glitchable with our measurement setup. We can also confirm that our choice for the coil and Z-Position enable glitching. We were able to find an easily glitchable area, with glitches resulting in faults potentially interesting for an attacker. Several measurements with the same parameters have to be taken to observe all possible glitch effects for a setting, because even though our setup repeatedly injected on a single position with the same parameters, different answers were obtained.

CHAPTER 7

Tracing

In this chapter, we explore what tracing can add for analyzing how glitches affect our target. First we explain how trace data can be read and then we analyze glitch results on the addsled and storesled test program.

7.1 Trace Data

When tracing is activated, trace data is stored in the ETB of the target. It can be specified for which address range a trace should be created. Our setup retrieves this data and parses it with etm2human [Shi]. An example how to read the decoded trace is given for listing 7.1.

```
1 trace flow started at 40303a8c, cycle 0
2 insn at 40303a8c: X cycle: 0 cond: PASS
3 insn at 40303a90: X cycle: 1 cond: PASS
4 insn at 40303a94: X cycle: 1 cond: PASS
5 insn at 40303a98: X cycle: 145 cond: PASS data_addr: 4804c194
```

Listing 7.1: An example trace decoded with etm2human demonstrating the basic capabilities

(1) (line 1 in listing 7.1) shows the start of the trace. It contains the address at which the trace collection started and the value of the internal cycle counter. (2) to (5) contain the address of the executed instruction, the cycle counter relative to the start of the trace collection and whether the instruction has passed its execution condition. Unconditional instructions are displayed as passed. Overall this trace contains in total four executed instructions, all of them either passed their condition code or had no condition.

It should be noted that the only real transfer of the instruction address is the trace beginning. All following instructions are assumed to be sequential by etm2human, even if they may not. The trace does not contain any information about whether the processor executed a branch to an address which should be predictable from reading the assembly code. (3) was executed 1 cycle

after (2). It should be noted that the absolute cycle value is not transferred. The trace data only contains packets in the form: "+8 cycles" "+8 cycles" "+3 cycles and 1 passed instruction". (4) was executed in the same cycle as (3). A technique known as Dual-Instruction enables executing two instructions in one cycle [p. "16-13" A8t]. (5) additionally contains a data address. Data addresses are only present for LSM instruction. The trace suggest that (5) is an LSM instruction, which also explains the relatively large amount of 144 cycles the instruction needed to execute.

7.2 Reliability of Tracing

To use tracing for the assessment of FI, we have to determine how reliable the trace data generation is during glitching.

We executed a testrun with the addsled program and the measurement setup with tracing. 32000 measurements were obtained with the Glitch-Intensity configured to 0 %, so that no glitch can occur. We collected the trace data parsed it and compared them against each other. An example trace is shown in listing 7.2. All data is similar in all measurements of the testrun, except the cycle values marked in red. The cycle offsets between instructions marked are sometimes more or fewer cycles. Although the instructions at (3), (15) and (19) are only a simple add instructions, they need several cycles to execute. We assume this behavior is caused by the Static Random Access Memory (SRAM) memory running on at a slower clock speed than the processor and the necessity to fetch new instructions from SRAM to cache from time to time.

```
1                       trace flow started at 40303a8c, cycle 0
2  movt R2, #0x4030     insn at 40303a8c: X cycle: 0 cond: PASS
3  str  R4, [R5]        insn at 40303a90: X cycle: 178 cond: PASS data_addr: 4804c194
4  add  R0, R0, #1      insn at 40303a94: X cycle: 178 cond: PASS
5  add  R1, R1, #1      insn at 40303a98: X cycle: 179 cond: PASS
6  add  R0, R0, #1      insn at 40303a9c: X cycle: 179 cond: PASS
7  add  R1, R1, #1      insn at 40303aa0: X cycle: 180 cond: PASS
8  add  R0, R0, #1      insn at 40303aa4: X cycle: 180 cond: PASS
9  add  R1, R1, #1      insn at 40303aa8: X cycle: 181 cond: PASS
10 add  R0, R0, #1      insn at 40303aac: X cycle: 181 cond: PASS
11 add  R1, R1, #1      insn at 40303ab0: X cycle: 182 cond: PASS
12 add  R0, R0, #1      insn at 40303ab4: X cycle: 182 cond: PASS
13 add  R1, R1, #1      insn at 40303ab8: X cycle: 183 cond: PASS
14 add  R0, R0, #1      insn at 40303abc: X cycle: 183 cond: PASS
15 add  R1, R1, #1      insn at 40303ac0: X cycle: 304 cond: PASS
16 add  R0, R0, #1      insn at 40303ac4: X cycle: 304 cond: PASS
17 add  R1, R1, #1      insn at 40303ac8: X cycle: 305 cond: PASS
18 add  R0, R0, #1      insn at 40303acc: X cycle: 305 cond: PASS
19 add  R1, R1, #1      insn at 40303ad0: X cycle: 327 cond: PASS
20 add  R0, R0, #1      insn at 40303ad4: X cycle: 327 cond: PASS
21 add  R1, R1, #1      insn at 40303ad8: X cycle: 328 cond: PASS
22 add  R0, R0, #1      insn at 40303adc: X cycle: 328 cond: PASS
23 add  R1, R1, #1      insn at 40303ae0: X cycle: 329 cond: PASS
24 mov  R4, #0x20000    insn at 40303ae4: X cycle: 330 cond: PASS
```

Listing 7.2: A decoded trace without glitch effects. The executed program is an addsled with 20 ADDs. The assembly is added on the right for convenience, but is not part of the actual trace data. Red cycle offsets are different between measurements in the same testrun.

We did a similar testrun with random Glitch-Intensity and Glitch-Offset values and compared them to the unaffected tracebuffer. Listing 7.3 shows a corrupted and glitched trace. Compared to the unaffected listing 7.2: It starts similar, continues with a corrupted part marked red in listing7.3, then continues normally but with the instruction address and cycle counter offset at a wrong value and finally ends abruptly with a jump to the exceptionvector. According to the UART answer of this measurement, the target encountered an Undefined Instruction Exception at address 0x40303ad8. The trace would perfectly match the UART answer, if the red part including the address and cycle offset was repaired. Comparing the raw glitched trace with the raw unaffected listing 7.4 shows several bit changes. The packet with hex representation BC is a cycle offset packet. Some of the BC instructions were interpreted as part of a branch packet 51 82 BC BC, although no branch occurred according the UART answer. Because of this we assume that glitching can cause bitflips in the tracebuffer. Also the next five randomly selected glitched measurements showed traces, best explained by tracebuffer corruption. The unreliability of the trace buffer has to be taken into account for further experiments.

```
trace flow started at 40303a8c, cycle 0
insn at 40303a8c: X cycle: 0 cond: PASS
insn at 40303a90: X cycle: 178 cond: PASS data_addr: 4804c194
insn at 40303a94: X cycle: 178 cond: PASS
insn at 40303a98: X cycle: 179 cond: FAIL
insn at 40303a9c: X cycle: 179 cond: FAIL
insn at cf313be4: X cycle: 284 cond: PASS
insn at cf313be8: X cycle: 284 cond: PASS
insn at cf313bec: X cycle: 285 cond: PASS
insn at cf313bf0: X cycle: 285 cond: PASS
insn at cf313bf4: X cycle: 307 cond: PASS
insn at cf313bf8: X cycle: 307 cond: PASS
insn at cf313bfc: X cycle: 308 cond: PASS
--->>> branching to 9f74f004 (Exception Vector: Undefined Instruction)
--->>> branching to 9f74f028
```

Listing 7.3: One of the decoded traces with glitch effects.

```
Glitched:
22 94 83 93 C0 04 | 92 | 8E | 51   B3   A1   82   BC   BC | BC | BC | BC | BC | BC | BC
 | BC | BC | BC | BC | BC | BC | BC | 82 | 82 | BC | BC | B0
Unaffected:
22 94 83 93 C0 04 | 92 | 82 | 82 | 82 | 82 | 82 | BC | BC | BC | BC | BC | BC | BC | BC
 | BC | BC | BC | BC | BC | BC | BC | 82 | 82 | BC | BC | B0
```

Listing 7.4: A part of an example decoded traces with glitch effects and the same part of a unaffected trace side by side. The glitch seems to have several corrupted bits, which leads to different decoded data and different interpretation of the packet boundary. Packet boundaries are indicted by a |, changed bytes by red color.

7.3 Tracing on with the storesled Test Program

We tried using tracing with a program containing LSM instructions, like the storesled test program. LSM are clearly distinguishable in traces and more verbose, so we assume we might tell more about glitch effects on LSM instructions than on non LSM instructions.

We executed a testrun with the storesled testprogram on a single highly glitchable location, derived from experiments in section 8.4. We obtained 8200 measurements with the Glitch-Intensity randomized between 40 % and 46 % and Glitch-Offset set to 6 ns.

100 % of the obtained abnormal results of this testrun have the expected value for R1 at the memory location for R0 and 00000000 at the memory location for R1. Listing 7.5 shows such an abnormal answer and the expected answer. Likewise listing 7.6 shows part of a decoded expected answer trace and a decoded abnormal answer trace. 0x00000000 is the value of an uninitialized memory location. The abnormal answer trace indicates that there was indeed no write to the memory location for R1, which explains the 00000000 value there. The register values read with OCD for R0, R1 and R2 are as expected, i.e. unaffected.

```
Expected Answer:
UART: NORMALEXECUTION In Memory: R0 ff00ffff  R1  ff01ffff  R2  ff02ffff...
OCD:  R0  ff00ffff  R1  ff01ffff  R2  ff02ffff...
Abnormal Answer:
UART: ABRMALEXECUTION In Memory: R0 ff01ffff  R1  00000000  R2  ff02ffff...
OCD:  R0  ff00ffff  R1  ff01ffff  R2  ff02ffff...
```

Listing 7.5: An unaffected/expected answer and a glitched/abnormal result. The registers have not changed according to the OCD readings, but the values stored in memory have changed.

```
Expected Answer:
...
insn at 40304484: X cycle: 444 cond: PASS
insn at 40304488: X cycle: 445 cond: PASS
insn at 4030448c: X cycle: 564 cond: PASS data_addr: 00306334 ;memory location for R0
insn at 40304490: X cycle: 564 cond: PASS
insn at 40304494: X cycle: 565 cond: PASS
insn at 40304498: X cycle: 684 cond: PASS data_addr: 00000038 ;memory location for R1
insn at 4030449c: X cycle: 684 cond: PASS
insn at 403044a0: X cycle: 685 cond: PASS
insn at 403044a4: X cycle: 804 cond: PASS data_addr: 4030633c ;memory location for R2
...
Abnormal Answer:
.... ;inconclusive data
insn at 40024040: X cycle: 614 cond: PASS
insn at 40024044: X cycle: 615 cond: PASS
insn at 40024048: X cycle: 731 cond: PASS data_addr: 00306334 ;memory location for R0
insn at 4002404c: X cycle: 731 cond: PASS
insn at 40024050: X cycle: 732 cond: PASS
insn at 40024054: X cycle: 847 cond: PASS data_addr: 4030633c ;memory location for R2
insn at 40024058: X cycle: 847 cond: PASS
insn at 4002405c: X cycle: 848 cond: PASS
insn at 40024060: X cycle: 967 cond: PASS data_addr: 00000040 ;memory location for R3
...
```

Listing 7.6: An unaffected/expected answer and a glitched/abnormal answer decoded trace. The instruction offsets of the abnormal answer are faulty, due to inconclusive data at the beginning of the trace buffer. The faulty data, might originate from trace buffer corruption.

We came up with several possible explanations, how the glitch might have affected the instruction execution:

The operand for the "str R0, [R12]" could be glitched to R1 ("str R1, [R12]") and additionally the store R1 instruction is skipped. Another possibility is that the store R1 instruction was glitched to write to the memory location for R0 instead, thus overwriting the R0 value. This again might be, because the mov instruction for setting the memory location was skipped or changed, resulting in the following instructions:

```
movw R12, memoryForR0
movt R12, memoryForR0
str R0, [R12]
---
---
str R1, [R12]
```

A further possibility is that during that the store execution was glitched into using a different location, however then "str R0, [R12]" should still be visible in the trace data if it has not been corrupted.

Unfortunately with the given information we cannot be certain of the glitch effect. However tracing and OCD enabled us to exclude some possibilities, because we see that indeed the value in R1 is the expected one and that there is indeed never a write to the memory location in R1.

7.4 Conclusions

We tried to use tracing for glitch effect analysis. Measurement time increase, the increased amount of data and the observed trace buffer corruption make tracing hard to use and prone to errors. Tracing did not help us understand the effects on programs without LSM instructions like the addsled test program. Tracing helped a little to understand glitches on the storesled, by excluding possible glitch effect explanations. An attacker or researcher should carefully judge whether tracing would add useful information to their test setup.

CHAPTER 8

Exceptions

This chapter describes the behavior observed while injecting faults into the nopsled, addsled, movsled, storesled, branchsled and comparesled test program. Because of the limited value of tracing as shown in chapter 7 and because our available research time was greatly limited, the fast measurement setup without tracing was used throughout this chapter. We show results, produced using our test programs with exception instrumentalization. Our technique gives an attacker insight in the operation of a target and could lead to reliably exploitable faults. Faults are usable, if the test program executes completely, but calculates a different result than in a non-faulty execution. Exceptions and full resets are faults aborting the normal execution and are usually not further usable for an attack. For each test program we created a:

whole surface testrun The whole surface testrun contains at least 100 measurements per location in a 50 by 50 grid over the processor surface. As parameters, unless otherwise mentioned, the Glitch-Offset was randomly set to values between 0 ns and 50 ns for each measurement and the Glitch-Intensity likewise to values between 40 % and 100 %. A surface scan can identify the sensitive areas against FI.

single location testrun Taking only 100 measurements per location is not necessarily enough to observe all interesting glitches and identify their probabilities and dependent parameters Therefore once a location with high usable glitch probability has been identified a testrun with at least 100,000 measurements for this location is created. Unless stated otherwise, the XY-Position was fixed and the Glitch-Offset and Glitch-Intensity were randomly set to values between 0 ns and 45 ns and 40 % and 100 % respectively for each measurement.

All remaining graphs exploring the parameter space are jittered scatter plots to prevent over-plotting. This means we add a small random offset (jitter) to each plotted dot, as otherwise dots would be plotted above each other.

8.1 Nopsled

To observe faults unrelated to specific instructions being executed, we use the nopsled. The nopsled's instructions (movne R0 R0) should not have any effect during unglitched execution and should not even pass its conditional execution requirements. We do not know, if the chosen nop instruction really does not influence the behavior while being glitched, but as expected we do observe significantly less abnormal behavior compared to the other intended to be more fault sensitive test programs.

Figures 8.1 and 8.2 show a testrun with the nopsled for the whole surface. We can observe scattered register corruptions, Undefined Instruction Exceptions, Data Abort Exceptions and Software Interrupt Exceptions. The huge amount of Undefined Instruction Exceptions concentrates on a distinct roundish area. The chip also has a sensitive area for Data Abort Exceptions. All Data Abort Exceptions were caused by instructions belonging to the exception handlers. The instruction causing the exceptions was determined by reading the exception handlers LR. We assume some of our exception handlers were glitched into causing a Data Abort Exception. Data Abort Exceptions can be caused by problems with the memory access. For example it could be that the processor can be glitched into an abnormal state, causing the exception handler to abort. Over 90 % of the register corruptions occurred adjacent or within the area with Undefined Instruction Exceptions. The area for no answer seems to correlate with the Undefined Instruction area. It seems the chip is only vulnerable against glitching in distinct areas.

A particular suprising observation is R15 corruption. R15 is the PC. If the PC really has a different value, it cannot reach the answering UART function anymore. Therefore it is more likely that only the storing of the R15 value was corrupted and not R15 itself.

Table 8.1 shows the overall distribution of answers for the single location testrun. The location used was 219095 motor steps in x direction and 243376 motor steps in y direction. The majority were unaffected results with 45.4758 % and Undefined Instruction Exceptions with 49.4401 %. The Undefined Instruction Exceptions might occur, because the instruction decoding is glitched and results in non-executable instructions. 0.6687 % of the measurements resulted in Data Abort Exceptions. All observed Data Abort Exceptions were caused by instructions being part of the exception handler.

Table 8.1: The answers obtained while injecting faults into the nopsled testprogram on a single location.

Occurrence rate	Measurement result
49.4401 %	Undefined Instruction Exception
45.4758 %	Expected Answer
4.2066 %	No Answer
0.6687 %	Software Interrupt Exception
0.1250 %	Data Abort Exception
0.0648 %	Abnormal Answer
Remaining %	Other

All the abnormal answers (0.0648 % of the measurement results) show register corruption. The three most dominant corrupted registers are R2, R6 and R0 with 0.0524 %, 0.0067 % and 0.0033 % of the measurements. The most dominant values for register corruptions are 0xfb3e6bff (0.0402

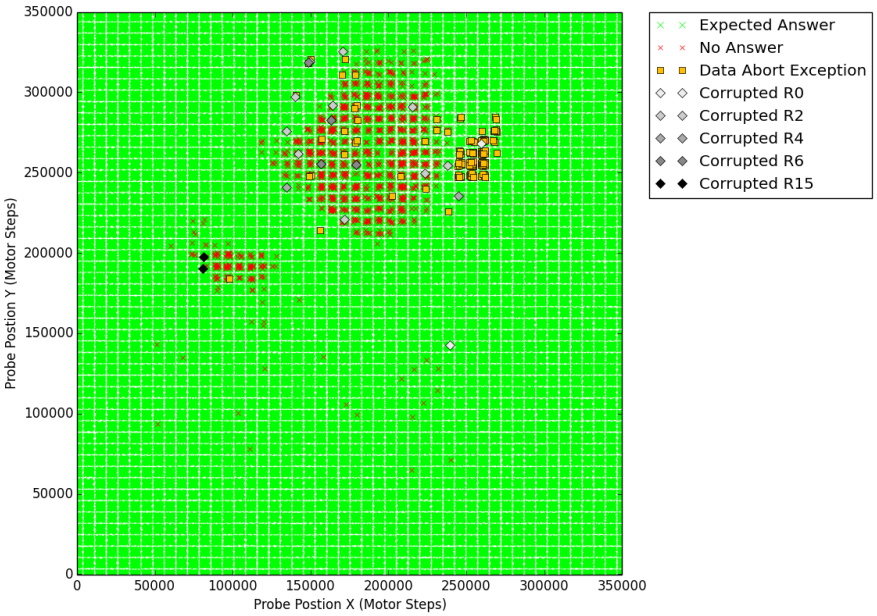

Figure 8.1: The surface explored while injecting into the nopsled test program. Only a few register corruptions and Data Abort Exceptions occurred. Undefined Instruction Exceptions and Software Interrupt Exceptions are excluded from the figure for better visibility to figure 8.2.

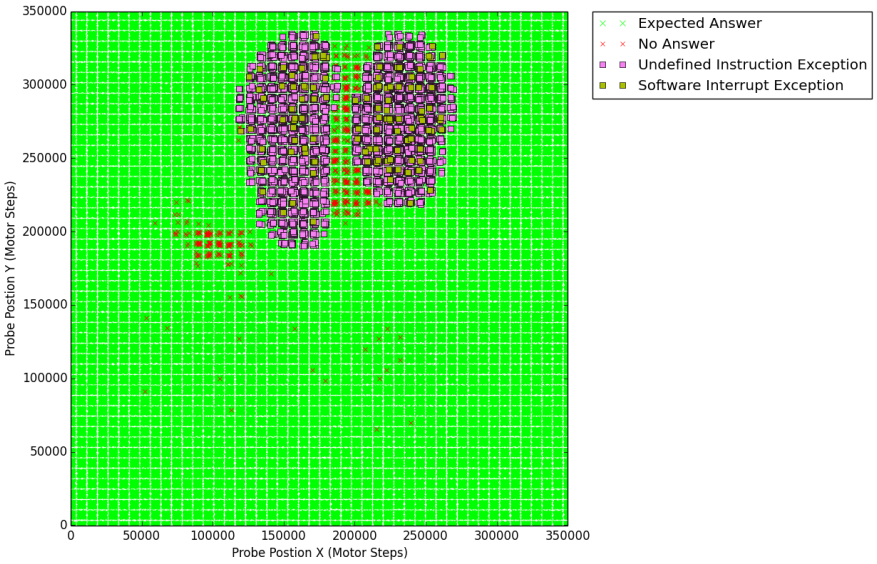

Figure 8.2: The parameter space explored while injecting into the nopsled test program. Undefined Instruction Exceptions and Software Interrupt Exceptions form the majority of glitch effects. Data Abort Exceptions and register corruptions are excluded from the figure for better visibility and put into figure 8.2.

%) and 0xecf9afff (0.0123 %) for R2 and 0xecf9afff(0.0056 %) for R6. The parameters causing the register corruption are illustrated in figure 8.3. Undefined Instruction Exceptions and Software Interrupt Exceptions occur in the whole parameter space and are excluded from the figure for better readability. The in the figure seemingly empty parameter space contains only exceptions and no expected answers.

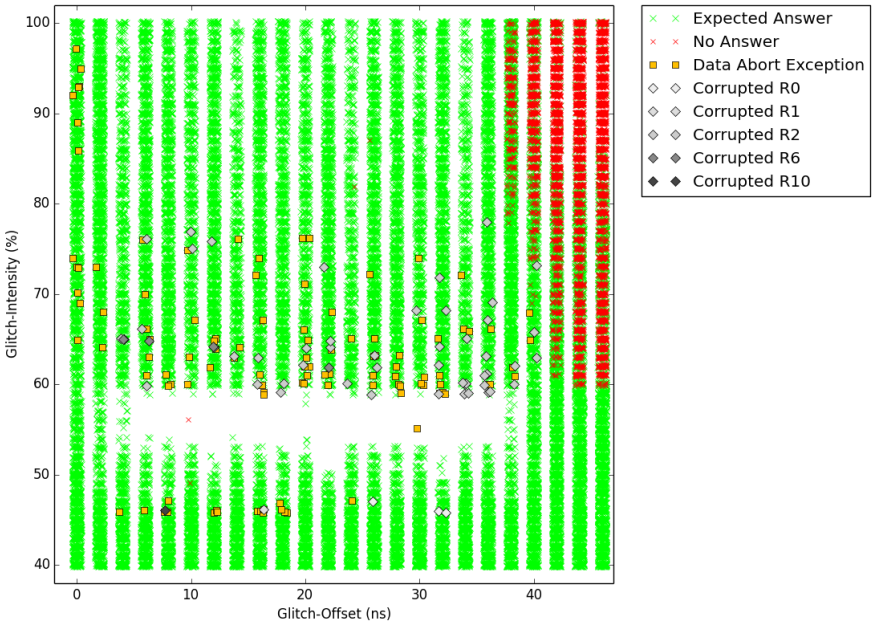

Figure 8.3: The parameter space explored while injecting into the nopsled test program. Only a few register corruptions and Data Abort Exceptions occurred. Undefined Instruction Exceptions and Software Interrupt Exceptions occur in the whole parameter space and are excluded from the figure for better readability.

Figure 8.4 shows the relation between the Glitch-Offset and the LR address. There is a clear relation between a later injection and a higher instruction address, which was reported as cause for the exception. Usually two instructions are executed in the same clock cycle, which explains why every second address has nearly no exceptions. For example column 28 ns Glitch-Offset contains exceptions for four different addresses. We conclude that our measurement setup has the ability to glitch with a precision of roughly four clock cycles and/or that a single instruction is can be affected in several clock cycles.

In conclusion our target is also glitchable while executing "MOVNE R0, R0", which should behave like a nop instruction. We were still able to observe register corruption interesting for an attacker, but as we will see later with a low probability compared to the experiments on other test programs. Exception logging enabled us to roughly estimate the precision of our measurement setup.

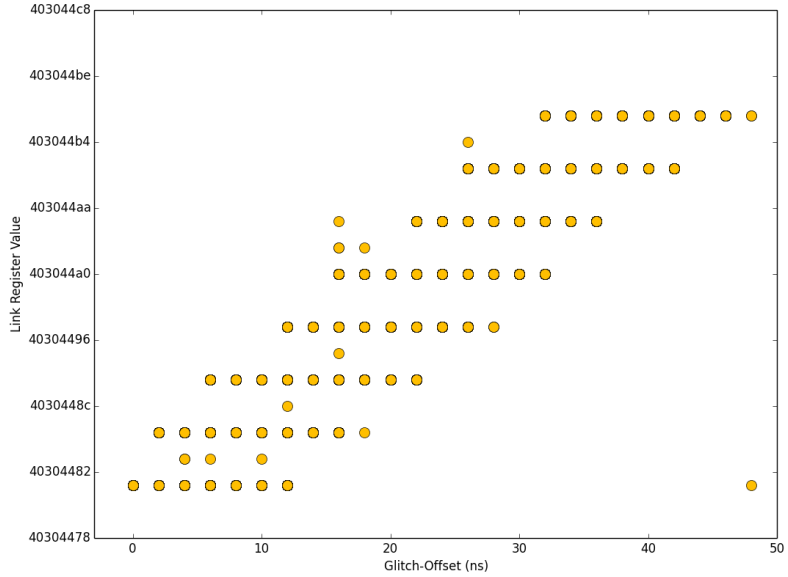

Figure 8.4: The nopsled single location testrun exception are plotted as yellow dots. The X-axis shows the Glitch-Offset after the trigger signal, before the injection was performed. The Y-axis shows the LR address stored by the exception handler.

8.2 Addsled

To observe glitches related to specific instructions we created different test programs. E.g. in this section we analyze the effects of EMFI on add instructions using the addsled.

Figures 8.5 and 8.6 show a testrun with the addsled for the whole surface. We can observe similar occurrence areas for exceptions and successful glitches. Apart from a few scattered register corruptions, Undefined Instruction Exceptions and Software Interrupt Exceptions, just like in the nopSled testrun, we observe far more Data Abort Exceptions and additionally Prefetch Abort Exceptions in figure 8.5. Figure 8.6 shows the effects on R0 and R1, the two registers used in the add instructions. We observe two areas: One where the addsled counts to too small values and one where the addsled results in too high values almost exclusively for R0.

Plotting the Glitch-Intensity versus the Glitch-Offset for the whole surface in figure 8.7 reveals that these location-wise separate areas also differ for the remaining parameters. The too low R0 and R1 faults occur exclusively around 40 to 50 ns Glitch-Offset and above 70 % Glitch-Intensity. For Undefined Instruction Exceptions, it is checked and then plotted, whether the causing instruction is one of the three arbitrarily selected instruction addresses in the addsled. These symbols were plotted at 32, 34, and 36 % Glitch-Intensity, but the original exceptions Glitch-Intensity values were ignored. This shows that an instruction can only be glitched to cause an Undefined Instruction Exception within a certain time period or these time ranges are caused by our setups imprecision. These time periods are several cycles long, possibly because

the instruction is already in a cache or in the pipeline. The time periods that can be affected for different instructions overlap, so an attack cannot be sure to influence a certain instruction precisely.

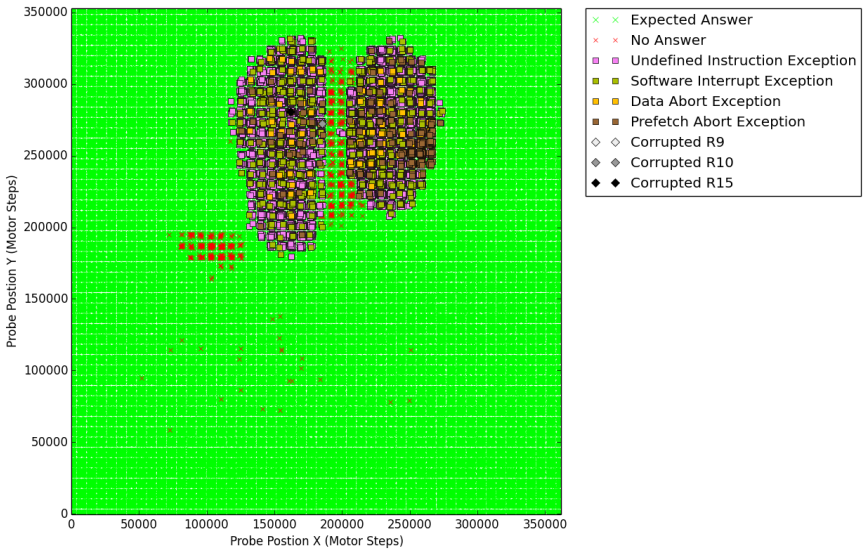

Figure 8.5: The surface explored while injecting into the addsled test program. Apart from a few register corruptions, significant amounts of exceptions occurred. For better visibility, the effects on R0 and R1 are shown in figure 8.6.

Figures 8.8 and 8.9 show parts of figure 8.7, separated into the left and right of the two roundish glitch areas in figures 8.5 and 8.6. The exact separation location is 200,000 motor steps in x direction. We observe that the whole picture for too high R0 instructions and Undefined Exception shifts in time. Therefore the same instruction is influenceable at different distinct locations at different times. This might be due to the instruction flow literally going from right to left through the board. Unfortunately the position and physical layout of the pipeline is unknown to us. Also decapsulation the processor in appendix A couldn't confirm our assumption.

The parameter exploration from the whole surface scan revealed such precise parameters that we configured the single location testrun to only take 1000 measurements with Glitch-Intensity and Glitch-Offset between 45ns and 50ns and 95 % and 100 %. Table 8.2 shows the distribution of answers. In abnormal answers the received values for both registers were 9 instead of 10. This could be due to a single instruction skip glitch. The 73.0807 % of always similar abnormal answers, presents attackers with a high-probability and predictable attack vector. Instead of each add instruction adding 1, different amounts could be added e.g. 1, 2, 4, 8 ..., making it obvious which the precise instruction was skipped by viewing the result. We leave this experiment for future work.

In conclusion our parameter exploration approach was able to find a reliable and reproducible glitch, behaving like an instruction skip for add instructions. This is an indication that our approach is valid for systematically exploring new targets with the goal of finding attack vectors.

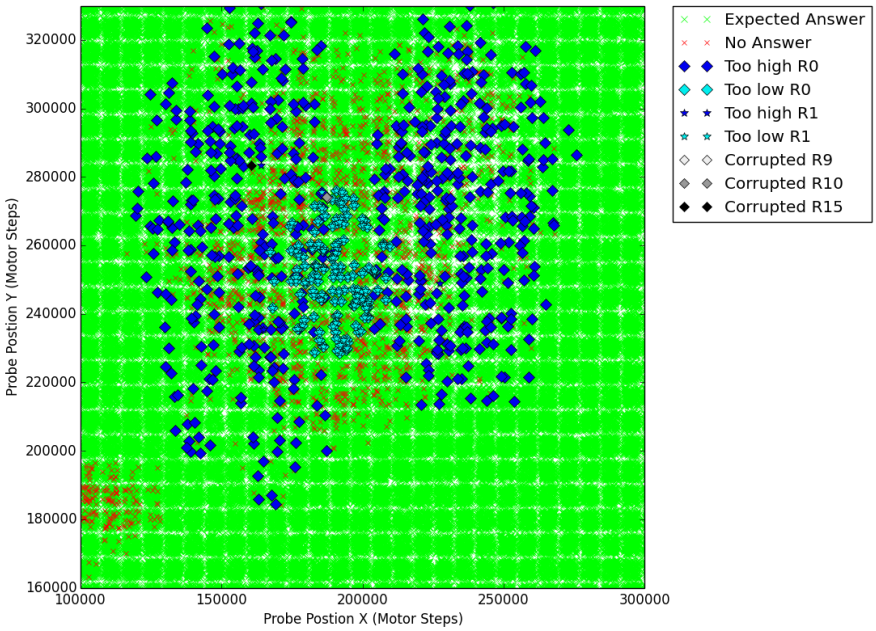

Figure 8.6: The surface explored while injecting into the addsled test program. The figure is focused on the sensitive area. Two register corruptions areas can be observed: Too low R0 and R1 together or a too high R0. For better visibility exceptions are excluded here and visible in figure 8.5.

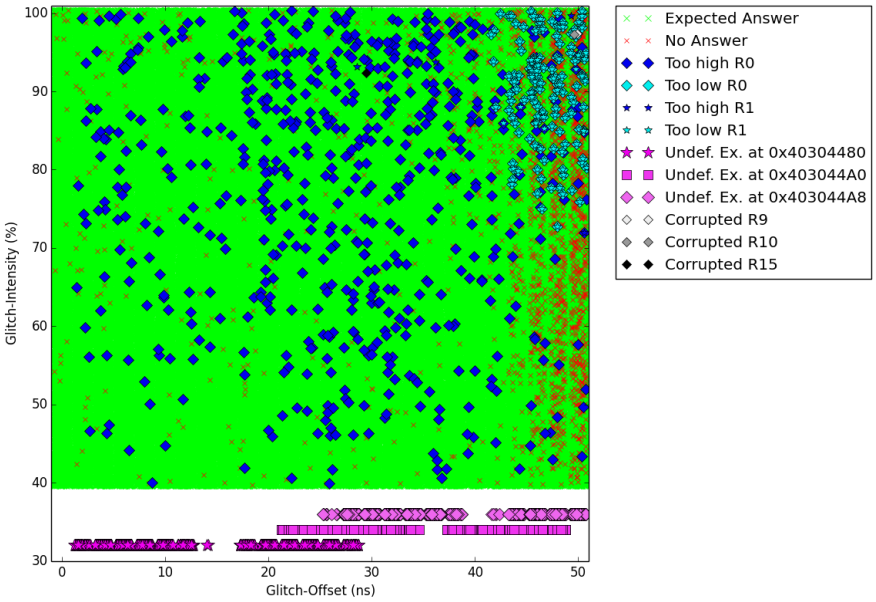

Figure 8.7: The parameter space of the whole surface testrun. Exceptions are not drawn for better readability. The location-wise different faults also have different glitch parameters. The pink Undefined Instruction Exceptions caused by the selected addresses are only plotted with the correct Glitch Offset, the Glitch-Offset are set manually.

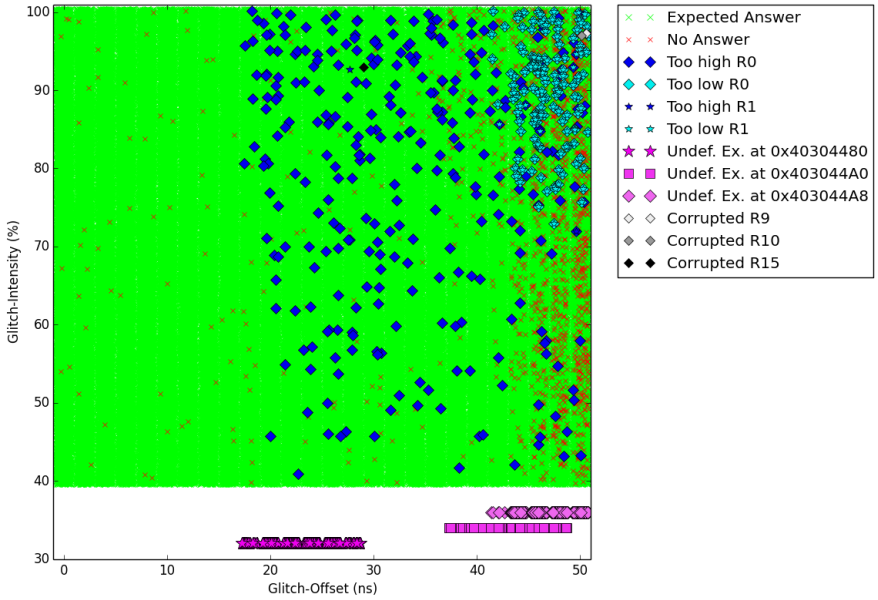

Figure 8.8: Same as figure 8.7, but only for measurements taken between 0 and 200000 motor steps on the X-Axis of the measurement setup.

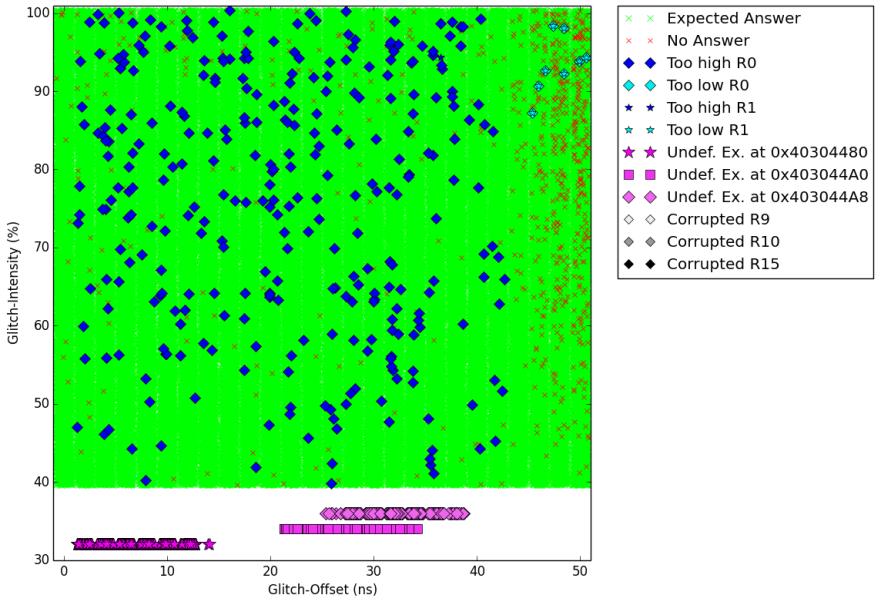

Figure 8.9: Same as figure 8.7, but only for measurements taken above 200000 motor steps on the X-Axis of the measurement setup.

Table 8.2: The answers obtained while injecting faults into the nopsled testprogram on a single location.

Occurrence rate	Measurement result
73.0807 %	Abnormal Answer
20.339 %	Expected Answer
6.5803 %	No Answer
0.000 %	Exceptions

8.3 Movsled

Figure 8.10 shows the whole surface and figure 8.11 show a single location testrun with the movsled test program. Again we see distinct areas for certain types of faults and a high correlation between the sensitive area for exceptions and the sensitive area for usable faults. The only usable fault occurring in the single location testrun is R0 corruption. This indicates that a "mov R0, R0" is affected in a way that it changes the value in R0. Follow up measurements with the same instruction, but another register could create more insights on this fault, but had to be omitted due to time reasons.

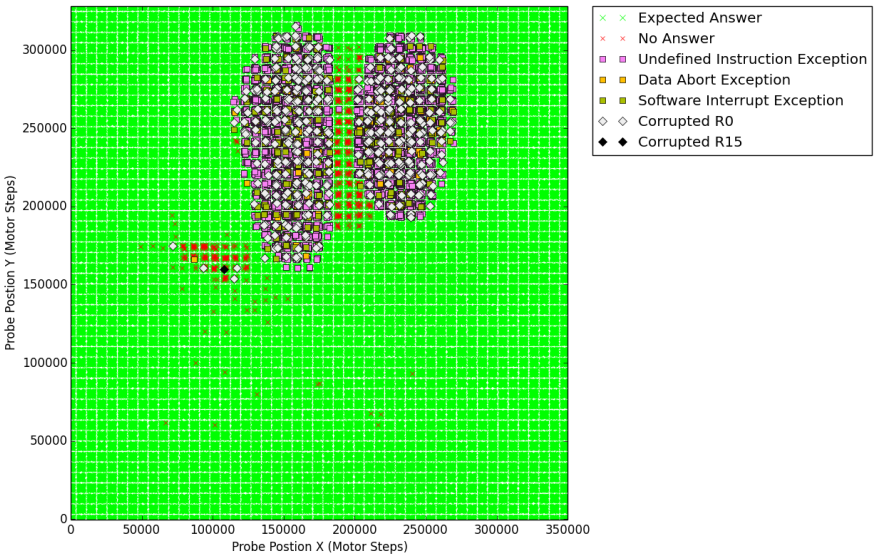

Figure 8.10

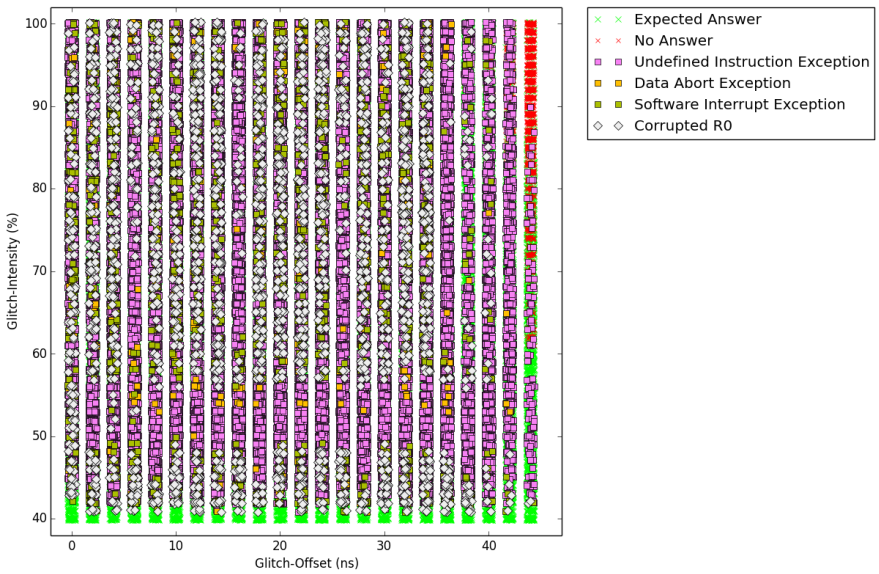

Figure 8.11: A single location's parameter space explored while injecting into the movsled test program.

8.4 Storesled

Figure 8.12 shows a whole surface testrun performed on the storesled. Faults for half of the registers were observed in the time period of only the first 45 ns Glitch-Offset. The single location testrun in figure 8.13 reveals that each store instruction has precisely targetable, partially with other store instructions overlapping, parameter areas. There is a clear relation between a later used register in the program code and a timely later register corruption in the figure. The presence of a store instruction seems to correlate with a higher amount of Data Abort Instructions. Table 8.3 shows a higher amount of Data Abort Exceptions than e.g. the movsled, addsled and nopsled.

Table 8.3: The answers obtained while injecting faults into the storesled test program on a single location.

Occurrence rate	Measurement result
38.2350 %	Undefined Instruction Exception
34.3510 %	Expected Answer
12.2340 %	Abnormal Answer
10.2500 %	Data Abort Exception
3.4210 %	No Answer
Remaining %	Other

We repeated the single location testrun for another highly glitchable location with a longer Glitch-Offset range of 0 to 150 ns, to see how it affects the later store instructions not visible in the up to 45ns testrun. A different location was used, because with our setup we are not able to precisely find the same location again, if the setup is moved. Figure 8.14 shows the results. The relation between Glitch-Offset and stored instruction seems to be only valid for the first

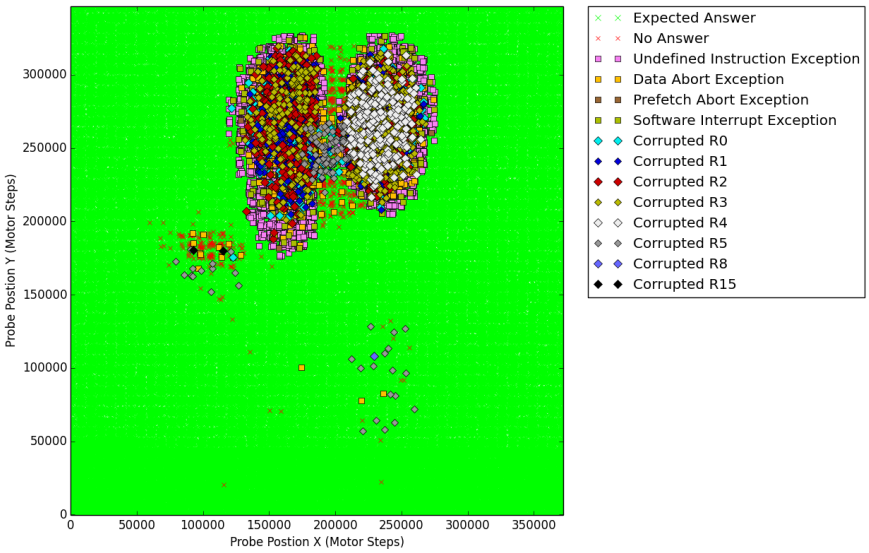

Figure 8.12: The surface explored while injecting into the storesled test program. Some registers have a specific glitchable region; for other registers the glitchable areas are distributed more evenly throughout the total glitchable area.

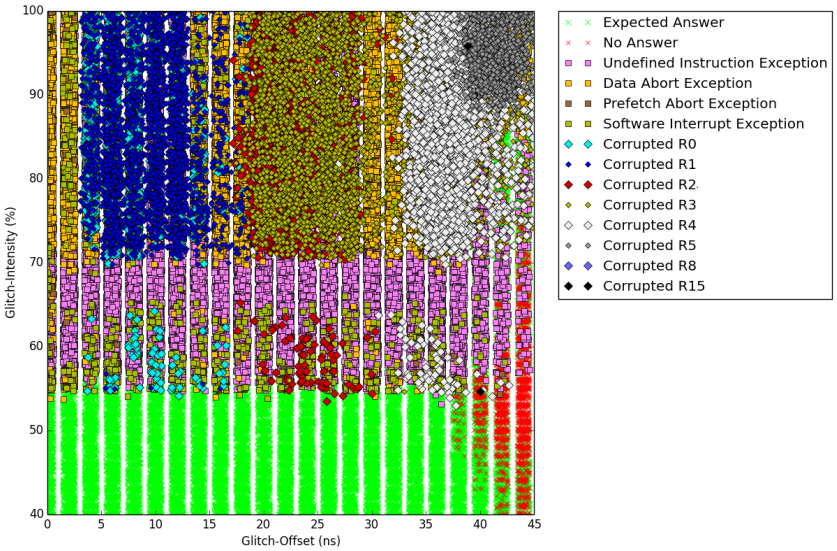

Figure 8.13: A single location's parameter space explored while injecting into the storesled test program. There are clear parameter regions for the six registers.

few registers. Some register have precisely targetable ranges, others have only varying single successful glitch parameters. The scarce results after 50 ns, might be due to the whole surface testrun being performed for up to 50 ns.

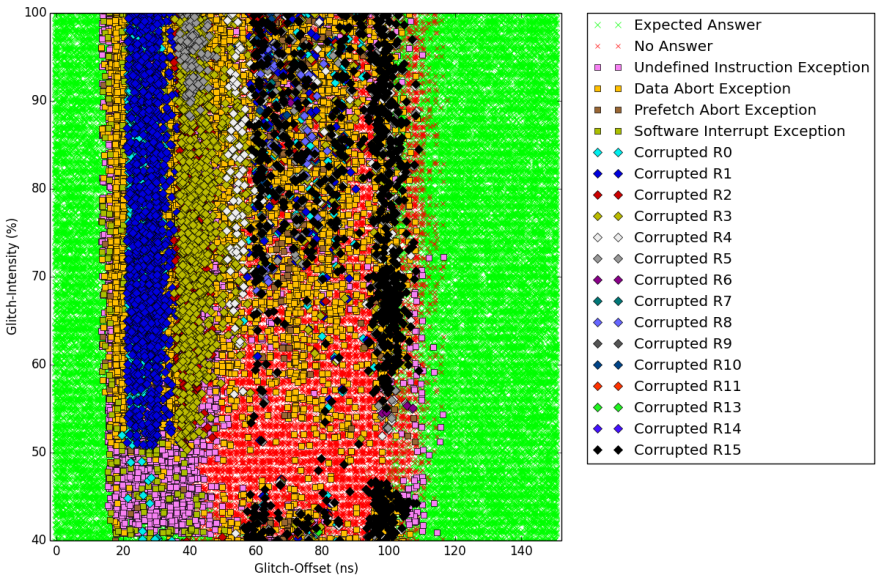

Figure 8.14: The parameter space explored while injecting into the storesled test program on a different location than figure 8.13. The clear regions for registers and relation between later glitches and higher registers from figure 8.13 does not hold for Glitch-Offsets above 60 ns.

The relation between the LR values in Data Abort Exceptions and the Glitch-Offset shows a partial trend between lower instruction addresses (i.e. earlier store instructions and lower registers) and a earlier Glitch-Offset. This relation seems to vanish after the first 50 ns.

In conclusion it is challenging to find reasonable explanations for the fault patterns in the storesled test program. However the experiment shows clear parameters for glitching the individual store instructions.

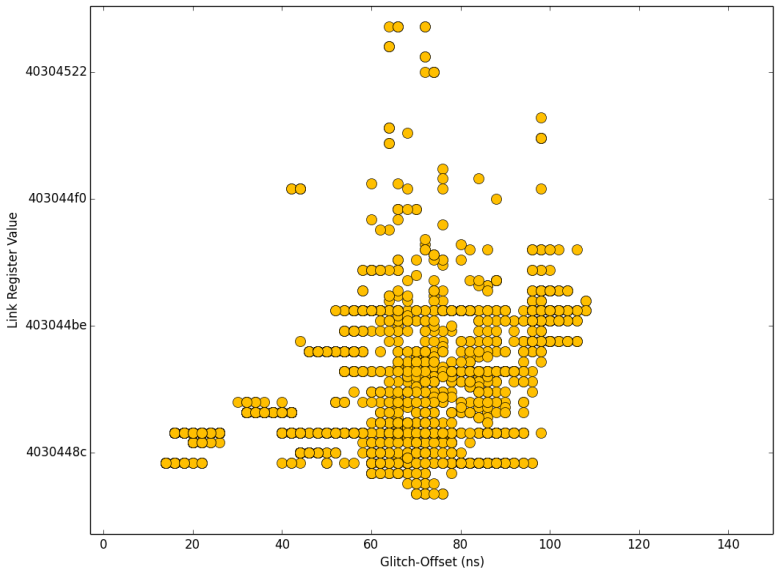

Figure 8.15: The exceptions of a testrun with the storesled test program and the Glitch-Offsets between 0 to 150 ns are plotted as yellow dots. The X-axis shows the Glitch-Offset waited after the trigger signal before the injection was performed. The Y-axis shows the LR address stored by the exception handler.

8.5 Branchsled

Figure 8.16 shows a whole surface testrun performed on the branchsled. Apart from exceptions and a few R15 corruptions no faults were observed (especially no usable glitches).

Because of the absence of any register corruptions, we conclude that glitching the branchsled is not producing any usable faults with our measurement setup and selection of parameters.

Because no usable glitches were observed, we refrained from performing a single location testrun.

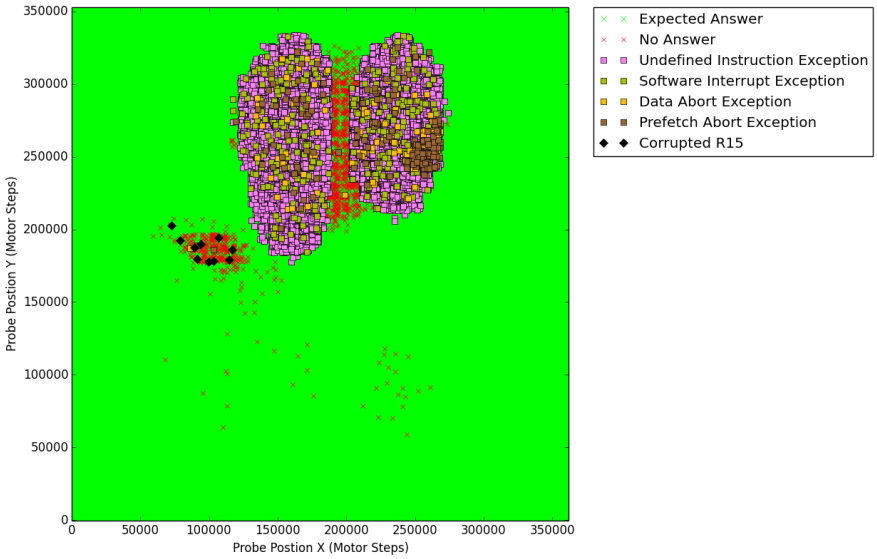

Figure 8.16: The surface area explored while injecting into the branchsled test program. Only exceptions and a few R15 corruptions occurred.

8.6 Comparesled

Figure 8.17 shows the whole surface scan for the branchsled test program. We derived the most sensitive location (177104 motor steps in x direction and 302021 motor steps in y direction) for abnormal answers for the single location testrun in figure 8.18.

Table 8.4 shows the overall distribution of answers for the single location testrun. Two abnormal answers were observed. One form of abnormal answer contains the unchanged initial R0 as the result, which could mean that the compare and branch combination being glitched into a branching state. The second form of abnormal answer contains the still initial R0 and R1 values, which should normally be modified.

Table 8.4: The answers obtained while injecting faults into the comparesled test program on a single location.

Occurrence rate	Measurement result
71.5409 %	Expected Answer
23.2388 %	Undefined Instruction Exception
2.4756 %	No Answer
0.9984 %	Software Interrupt Exception
0.5649 %	Abnormal Answer initial R0
0.4110 %	Abnormal Answer initial R0 and R1
Remaining %	Other

In conclusion some abnormal answers observed could refer to a branch being taken. However the second abnormal answer occurring is not easily explainable.

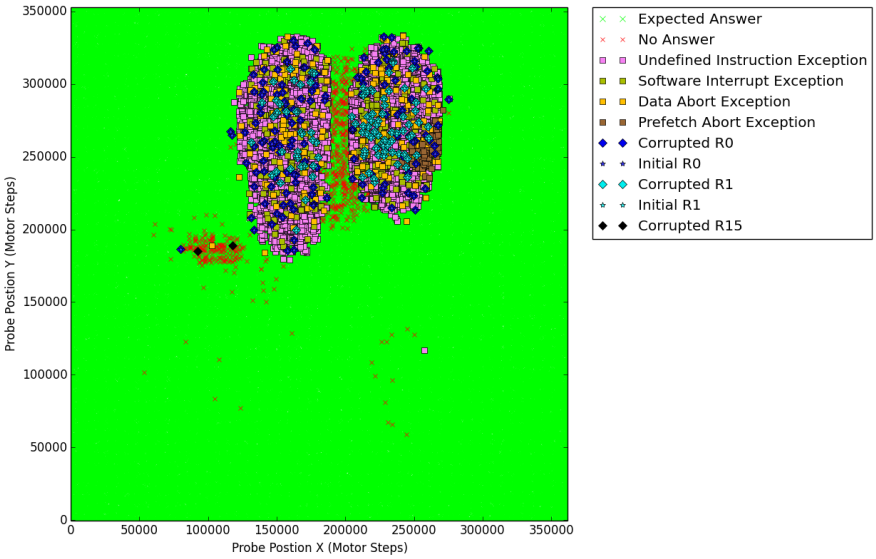

Figure 8.17: The surface explored while injecting into the comparesled test program. We observed exceptions, a few R15 corruptions, as well as several glitches, which returned the unchanged initial values for R0 or for R0 and R1 as result.

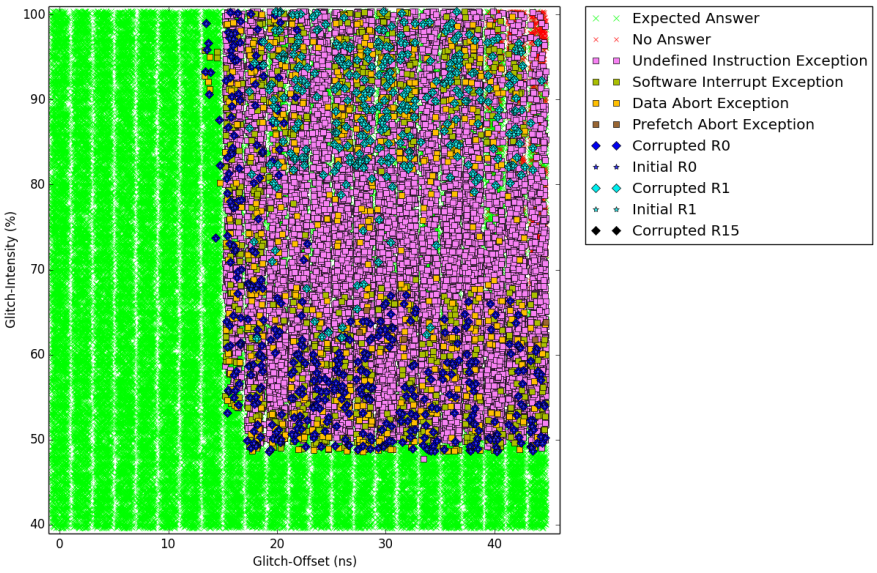

Figure 8.18: The parameter space explored while injecting into a single location in the branchsled test program. We observed exceptions, a few R15 corruptions, as well as several glitches, which returned the unchanged initial values for R0 or for R0 and R1 as result.

8.7 Conclusion

By injecting into our test programs and systematically exploring our parameter space, we were able to inject faults usable for an attacker. Table 8.5 summarizes our findings. We were able to find reliable reproducible glitches, which could be critical for security.

Table 8.5: The findings while injecting into the different test programs

Test program	Findings
Nopsled	Only a quite small amount of register corruption and thereby usable glitches compared to the other test programs
Addsled	Clear usable fault regions for an attacker. Our method found a location with above 70 % usable fault probability. Fault might be instruction skip.
Movsled	Clear usable fault regions for an attacker.
Storesled	Plenty of usable faults for an attacker. Partially clear relation between injection time and instruction sequence in the source code, partially inconclusive results. Clear regions for some fault types.
Branchsled	Nearly no usable faults, seems similar to the nopsled
Comparesled	May usable faults, might have glitched the compare statement

We showed that a combination of test programs while using exceptions and reading out register values helps to understand the glitch effects and assess their usability and reproducibility.

We saw that reading out registers is highly valuable for observing glitch effects. It is a fast and easy to implement technique. We cannot deduce how exactly a glitch affected the instruction execution, but for an attacker knowing that register values can be modified or that instructions can be skipped is enough for an attack. We showed that different test programs give different results, which could be an indication how individual instructions can be glitched. By gradually narrowing down the parameter space, this technique enabled us to find glitches occurring with above 50 % probability.

We saw that exceptions instrumentalization enabled us to distinguish between different types of exceptions instead of just having a not answering target. We are able to identify regions more sensitive to certain fault types. The exception address enabled us to show the instruction range we inject into and offers a method for testing the precision of our test setup. Despite that usually exceptions are not directly usable, they enable us to get a rough understanding of the faults occurring in the target. Reading out the link register values e.g.

Unfortunately exceptions are unusable glitches, because they mean the target aborted the normal execution.

CHAPTER 9

Summary, Conclusion and Future Work

In this thesis we presented and evaluated techniques for exploring glitch effects on a 32-bit high speed embedded device microprocessor's instruction execution. Chapter 1 and 2 introduced our research question and EMFI. Chapter 3 listed the different techniques, which can be used for observing faults caused by FI in state-of-the-art embedded device processors. We thereby provide researchers and security analysts with a list of techniques to choose from for their own experiments. From this list we selected several techniques to evaluate in experiments.

Chapter 4 and 5 introduced our measurement setup, target and the test programs. We presented several test programs with the intention to observe faults on specific instructions or groups of instructions.

Chapter 6 showed that faults are injectable into our target and that there are distinct chip surface areas, with high glitch sensitivity.

We introduced several test programs. Each test program was developed to cover one or more types of instructions. Experiments in chapter 8 showed that these test programs indeed show different fault behavior. The results give indications to which faults an instruction can be glitched. An attacker can use our provided or similar test programs to find the possible faults in a generic target and use them for crafting an attack.

We then explored the potential of the three selected glitch effect observation techniques: reading out registers with software, exceptions and tracing.

We concluded in chapter 8 that reading out registers is highly valuable for observing glitch effects. It is a fast and easy implemented technique compared to tracing. We cannot deduce how exactly a glitch affected the instruction execution, but for an attacker knowing that register values can be modified or that instructions can be skipped , it is enough for an attack. By gradually narrowing down the parameter space, this technique enabled us to find faults occurring with above 50 % probability.

For some test programs it was easier to find a reasonable explanation for a faulty result, like instruction skip faults or changes in how an instruction is interpreted. On the other hand some test programs did not show any for an attacker usable faults. We thereby did show which of our proposed test programs were more suitable for exploring glitch effects than others.

Furthermore we saw in chapter 8 that exception instrumentalization enabled us to distinguish between different types of exceptions instead of just having a not answering target. We were able to identify regions more sensitive to certain fault types. The exception cause addresses enabled us to show the instruction range we inject into and offers a method for testing the precision of our test setup. Unfortunately exceptions are unusable glitches, because they mean the target aborted the normal execution.

Unfortunately tracing used in chapter 7 did not provide additional usable information in most cases. On the contrary, tracing adds a lot of time overhead, while being prone to arbitrary trace data corruption. We saw some benefit, knowledge of the used memory addresses, for LSM instructions, but the trade-off between overhead and information gain should be carefully considered.

By using the mentioned techniques and test programs, we were able to cause several potentially security critical faults for a previously completely unexplored target. We also showed that EMFI can be used for a high speed target running with a clock frequency of 1 Ghz.

We indeed managed to explore the glitch effects on our target, but our solution leaves room for improvement.

Future work could use the introduced techniques and apply them against a target with security countermeasures or try to create a systematical approach for aiding the process of finding a successful attack against a new unexplored target. We already outlined a systematical approach: select a test program, make a whole surface scan, select a highly sensitive location and inject there. For some testruns this approach led to faults with above 50 % probability. Maybe a more intelligent approach can be developed for not so straight forward cases.

The target programs selection could be improved and extended. For example in the addsled test program, instead of each add instruction adding 1, different amounts could be added e.g. 1, 10, 100, 1000 ..., making it obvious which precise instruction was skipped by viewing the result. In our version we can assume only that, but not which instruction was skipped. Different typical instruction combinations could be evaluated, for example a compare and branch combination in the comparesled is such a typical pattern.

Future work could study the relationship between parameter space areas with exceptions and parameter space areas with faults usable for an attack. We already saw that usable faults occur at similar locations or closely to exceptions, but maybe there is general rule where to or where not to search for usable faults if we already know the exceptions.

Bibliography

[Aar13] Maurice Aarts: 'Electromagnetic Fault Injection using Transient Pulse Injections'. MA thesis. Technical University Eindhoven, 2013 (cit. on pp. 2, 6, 9).

[Ami06] Frederic Amiel, Christophe Clavier, Michael Tunstall, Avenue Des Jujubiers, La Ciotat, Smart Card Centre, and Information Security Group: 'Collision fault analysis of DPAresistant algorithms, Fault Diagnosis and Tolerance'. In *Cryptography 2006 — FDTC 06*. Springer-Verlag, 2006: pp. 223–236 (cit. on pp. 1, 5).

[Exc] *Application Note 25 Exception Handling on the ARM*. ARM DAI 0025E. 1996 (cit. on pp. 10, 11).

[Bar09] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi: 'Low Voltage Fault Attacks on the RSA Cryptosystem.' In *FDTC*. IEEE Computer Society, 2009: pp. 23–31 (cit. on pp. 1, 5).

[Bon01] Dan Boneh, Richard A. Demillo, and Richard J. Lipton: 'On the Importance of Eliminating Errors in Cryptographic Computations'. In *Journal of Cryptology* (2001), vol. 14: pp. 101–119 (cit. on p. 1).

[BVa] Riscure BV: *EMFI tranient probe Datasheet*. URL: https://www.riscure.com/documents/datasheet_em-fi_transient_probe.pdf (visited on 04/23/2014) (cit. on p. 24).

[BVb] Riscure BV: *VC Glitcher Datasheet*. URL: https://www.riscure.com/documents/datasheet_vcglitcher.pdf (visited on 04/23/2014) (cit. on p. 25).

[A8t] *Cortex™-A8 Technical Reference Manual Revision: r3p2*. ARM DDI 0344K (ID060510). 2010 (cit. on pp. 10, 11, 32).

[Cou] *Cortex™-M3 Revision r2p0 Technical Reference Manual*. ARM DDI 0337H. 2010 (cit. on p. 13).

[Deh13] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria: 'Electromagnetic Glitch on the AES Round Counter'. In *Constructive Side-Channel Analysis and Secure Design*. Vol. 7864. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013: pp. 17–31 (cit. on pp. 2, 25).

[Deh12a] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria: 'Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES'. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*. IEEE, 2012: pp. 7–15 (cit. on p. 2).

[Deh12b] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, P. Orsatelli, Philippe Maurine, and Assia Tria: 'Injection of transient faults using electromagnetic pulses -Practical results on a cryptographic system-.' In *IACR Cryptology ePrint Archive* (2012), vol. 2012. informal publication: p. 123 (cit. on pp. 2, 9).

[Etm] *Embedded Trace Macrocell™ ETMv1.0 to ETMv3.5 Architecture Specification*. ARM DAI 0025E. 2011 (cit. on p. 12).

[EMV] LLC EMVCo: *EMV Security Guidelines, EMVCo Security Evaluation Process*. URL: http://www.emvco.com/download_agreement.aspx?id=393 (visited on 05/23/2014) (cit. on p. 1).

[Eng] DENX Software Engineering: *The DENX U-Boot and Linux Guide (DULG) for canyonlands*. URL: http://www.denx.de/wiki/DULG/Manual (visited on 04/23/2014) (cit. on p. 26).

[FC14] Jacques Fournier Franck Courbon Philippe Loubet-Moundi and Assia Tria: 'Adjusting laser injections for fully controlled faults'. In *in the Proceedings of COSADE 2014* (2014), vol. (cit. on p. 2).

[Gmb] Lauterbach Gmbh: *ARM-ETM Training*. URL: http://www.lauterbach.com/pdf/training_arm_etm.pdf (visited on 04/23/2014) (cit. on pp. 12, 16, 17).

[Gov03] Sudhakar Govindavajhala and Andrew W. Appel: 'Using Memory Errors to Attack a Virtual Machine'. In *IEEE Symposium on Security and Privacy*. 2003: pp. 154–165 (cit. on pp. 1, 5).

[Hsu97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer: 'Fault Injection Techniques and Tools'. In *Computer* (Apr. 1997), vol. 30(4): pp. 75–82 (cit. on p. 1).

[Ins] Texas Instruments: *Sitara ARM Cortex-A8 Microprocessor Product Description*. URL: http://www.ti.com/product/am3358 (visited on 04/23/2014) (cit. on pp. 15, 68).

[Kar95] Johan Karlsson and Peter Folkesson: 'Application of three physical fault injection techniques to the experimental assessment of the MARS architecture'. In. IEEE Computer Society Press, 1995: pp. 267–287 (cit. on p. 6).

[Lim] ARM Limited: *NOP ARM pseudo-instruction*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0170b/Caccegih.html (visited on 04/23/2014) (cit. on p. 18).

[Mau12] Philippe Maurine, Karim Tobich, Thomas Ordas, and Pierre Yvan Liardet: 'Yet Another Fault Injection Technique : by Forward Body Biasing Injection'. Anglais. In *YACC'2012: Yet Another Conference on Cryptography*. Porquerolles Island, France, Sept. 2012 (cit. on p. 1).

[Mor13] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz: 'Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller'. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*. 2013: pp. 77–88 (cit. on pp. 2, 9, 10, 12, 13, 17, 26, 27).

[Ope] *Open On-Chip Debugger General Commands*. URL: http://openocd.sourceforge.net/doc/html/General-Commands.html (visited on 04/23/2014) (cit. on p. 11).

[Orl03] J. Orloff, L. Swanson, and M. Utlaut: *High Resolution Focused Ion Beams: Fib and Its Applications : The Physics of Liquid Metal Ion Sources and Ion Optics and Their Application to Focused Ion Beam Technology*. Springer US, 2003 (cit. on p. 6).

[RV13] Jasper van Woudenberg Rajesh Velegalati Robert Van Spyk: 'Electro Magnetic Fault Injection in Practice'. In (2013), vol. (cit. on p. 2).

[Sch08] J. Schmidt and C. Herbst: 'A Practical Fault Attack on Square and Multiply'. In*Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC '08. 5th Workshop on*. Aug. 2008: pp. 53–58 (cit. on p. 1).

[Shi] Alexander Shishkin: *etm2human decoder for trace data output by Embedded Trace Macrocells*. URL: https://github.com/virtuoso/etm2human (visited on 04/23/2014) (cit. on pp. 12, 31).

[Sko05] Sergei P. Skorobogatov: *Semi-invasive attacks – A new approach to hardware security analysis*. 2005 (cit. on pp. 1, 5, 6).

[Spr13] Albert Spruyt: 'Building fault models for microcontrollers'. In*Workshop on Trustworthy Manufacturing and Utilization of Secure Devices (TRUDEVICE), Co-located with IEEE European Test Symposium (ETS)* (2013), vol. (cit. on pp. 9, 10).

[Ver11] I. Verbauwhede, D. Karaklajic, and J. Schmidt: 'The Fault Attack Jungle - A Classification Model to Guide You'. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*. Sept. 2011: pp. 3–8 (cit. on p. 1).

[Wou11] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini: 'Practical Optical Fault Injection on Secure Microcontrollers.' In *FDTC*. Ed. by Luca Breveglieri, Sylvain Guilley, Israel Koren, David Naccache, and Junko Takahashi. IEEE, 2011: pp. 91–99 (cit. on pp. 1, 6).

Abbreviations

A8 Cortex A8 microprocessor
AES Advanced Encryption Standard
AHB Advanced High Performance Bus
API Application Programming Interface
ARM Advanced RISC Machines
BBB Beagle Bone Black
CCS Code Composer Studio
EMFI Electromagnetic Fault Injection
EM Electromagnetic
ETB Embedded Trace Buffer
ETM Embedded Trace Macrocell
FI Fault Injection
FPGA Field Programmable Gate Array
HTM AHB Trace Macrocell
I/O Input/Output
IDE Integrated Development Environment
JTAG Joint Test Action Group
LR Link Register
LSM Load Store Multiple
OCD On Chip Debugger
OS Operation System
PC Program Counter
PLL Phase-Locked Loops
PoP Package on Package
RSARon Rivest, Adi Shamir and Leonard Adleman public-key cryptosystem
SIM Subscriber Identity Module
SRAM Static Random Access Memory
STB Set Top Boxes
TI Texas Instruments
UART Universal Asynchronous Receiver Transmitter

List of Figures

List of Tables

Listings

APPENDIX A

Decapsulated Beaglebone Processor

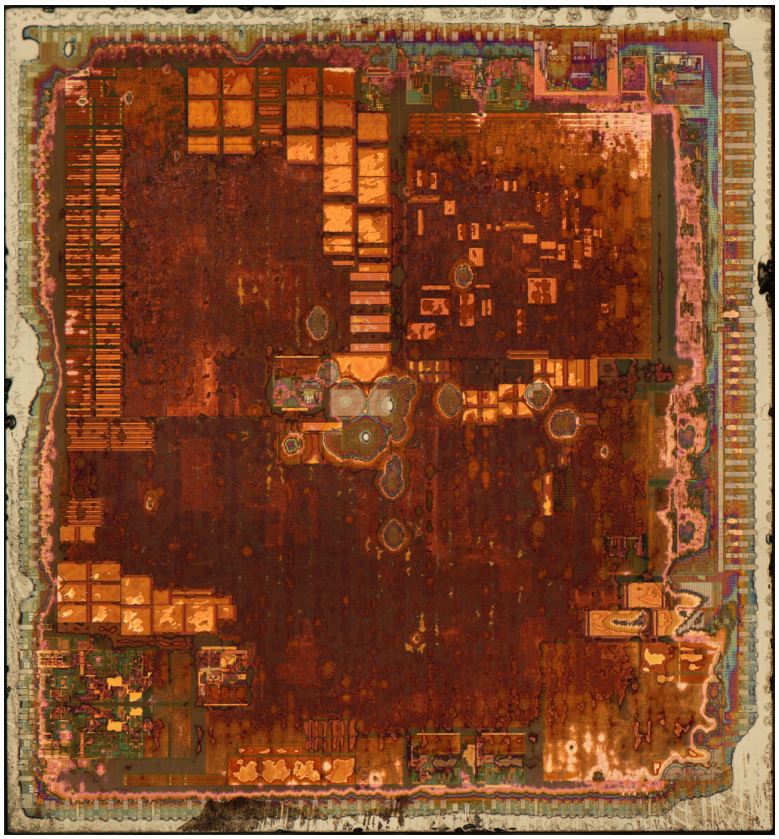

Figure A.1: The silicon layer of the BBB processor. The metal removal process left several impurities.

This chapter presents a view into the decapsulated BBB processor, the Texas Instruments Sitara

AM3358AZCZ100 [Ins]. Unfortunately due to our positioning setup, we cannot precisely tell where our setup injected faults successfully, because the chip images did not give us more than a rough impression of the chip layout. Therefore the decapsulation did not contribute to this thesis, but is still shown here for an interested reader.

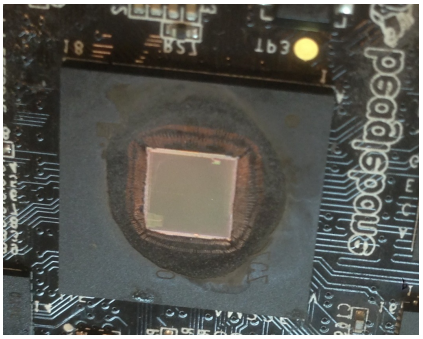

Figure A.2: The processor package opened up. The first metal layer of the die is visible.

Figure A.2 shows the opened BBB processor package. The first layer is a metal mesh. Figure A.3 shows a rough approximation where the die is during our experiments. For our experiments the injection probe reference positions are created by eyesight above the corners of the package. This positioning approach is sufficient for our experiments, but not for telling where exactly the die was during our experiment. It seems that the largest amount of glitches was injected not above the die, but above the bonding wires. Potentiality in our experiments most of the glitches are produced by currents in the bonding wires, instead of by injecting directly into the die.

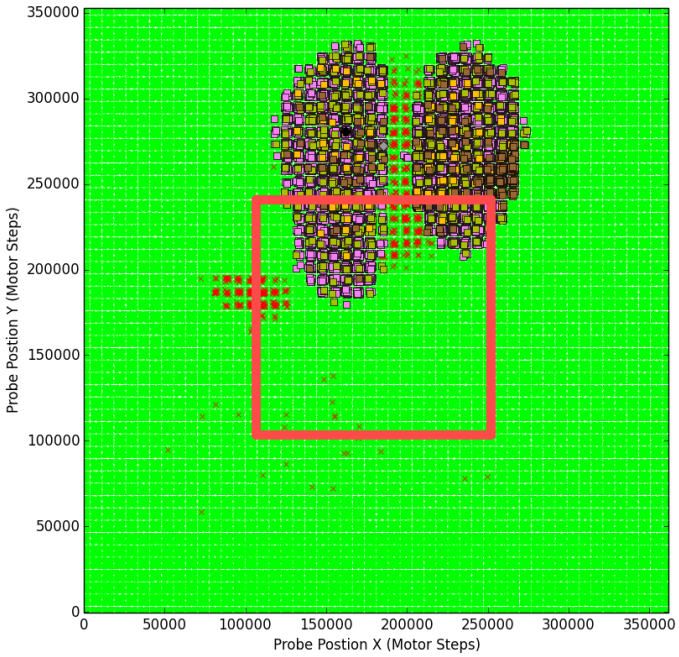

Figure A.3: Figure 8.5 with the approximate position of the die marked in red.

Figure A.1 shows a close-up view of the die with all metal layers removed. The quick and dirty

removal process left several marks and impurities. The rectangular brighter structures might be memory. In the top right corner we can see a separate logic unit, which might be the A8 processor.

All pictures in this chapter are oriented in the same way.