Building fault models for microcontrollers

Albert Spruyt

August 13, 2012

Abstract

Voltage glitching attacks can attain results that are logically not possible. The use of such attacks, is at times an opaque strategy. This paper presents a number of tests that can give insight into the effects that manifest themselves at the software level. Tests are presented for ALU-instructions, flow control instructions and memory instructions. These tests are performed on an XMEGA microcontroller and the results are used to create a fault model and an attack model. The fault model shows that, glitching causes faults in the instruction fetch phase. The faults are characterised by one bits turning into zero bits.

Contents

Chapter 1

Introduction

This paper is the result of the second research project for the System and Network Engineering master's program at the University of Amsterdam. The master's program has two separate tracks: networking and forensics. The research presented here was conducted for the forensics track.

The goal was to create a method to study the effects of voltage glitching on microcontrollers. Voltage glitching is a physical attack on integrated circuits. It works by varying the voltages supplied to a target. Glitching of integrated circuits allows some control over running code. For instance, to bypass protection measures.

Protection measures can range from PIN protection to full encryption. Forensic investigators are often interested in bypassing such measures. An example of such a case can be found in [1]. Moreover, the use of tools like JTAG is already a practice [2].

The aim was to create a tool that can give insight into glitches and allow these to be used more often. The insight gained in such a way could be used to create an attack model. This attack model can be used to perform attacks with the goal of extracting evidence.

The author would like to thank Cristofaro Mune and Niek Timmers from Riscure for their help and support.

Chapter 2

Background

Glitching attacks can accomplish things that logically cannot be achieved while attacking embedded systems. At the same time it is a prerequisite for many attacks to gain access to the code or obtain runtime control before other attacks (such as side channel analysis) can be applied.

These days, most common microcontrollers include features designed to protect the internal code from extraction preventing access to the code for further analysis. Code can be accessed via JTAG or bootloader interfaces, which can both be protected or disabled. It is expected that glitching will allow circumvention of these features. Examples of such microcontrollers are the MSP430 and the XMEGA. The project's focus will be on documenting exactly what the effects of voltage glitching are on the chip.

A number of papers describe generic methods for attacking chips; these include [3] and [4]. Research has been performed on Differential Fault Analysis (DFA), for instance by [5]. Less research has been done on the effects of voltage glitching. Voltage glitching is at times an opaque attack strategy. Systems can, and are, attacked using these methods but why the attacks work is at times poorly understood. Different faults can produce the same outcome. Looking at the assembly code of programs does not show the entire range of faults that can be abused. This project aims to create a fault model for subverting hardware and the software running on top of it.

Travis Goodspeed has previously conducted research on the MSP430 family and their boot loaders in [6] and [7]. These papers hint at the use of fault injection to bypass the password protection of these bootloaders. This differs from the research presented here. The goal of this research is not to bypass protection measures with glitches, but rather to create a model for understanding them.

An in-depth study of the effects of clock glitching on MEGA163 smart cards has been performed [8]. The study is closer to the spirit of this paper but differs on a number of points. The clock line is glitched, which is not possible on many embedded systems. Furthermore, the fault model is presented but the method cannot be easily discerned or transferred to a different microcontroller architecture.

A study of side channel attacks has been conducted on the XMEGA family's on-die cryptographic hardware [9]. The study focusses on the same microcontroller that will be targeted in this project. A key difference is the singular focus on the cryptographic instructions. This family of instructions will not be evaluated because they are not commonly available on microcontrollers.

Chapter 3

Approach

Due to the short duration of this project, the project approach had to be well defined. This chapter discusses the research questions, the project scope and their rationale.

3.1 Research questions

To help define the project's goal, a main question and a number of subquestions are defined. The goal is to create a method to model the impact of voltage glitches on microcontrollers. The main research question is formulated as:

What is the impact of faults injected into the power line of a microcontroller?

The main question is answered with the help of the following subquestions:

1. *Which methods or approaches can be used for voltage glitching?*
   This question will enumerate the different voltage glitching attacks.
2. *What are the different ways in which these faults or glitches affect operation of the microcontroller?*
   The methods enumerated for the previous question allow the introduction of faults in microcontrollers. Before tests can be created to determine the effects of faults, we will first detail the operation of microcontrollers. In addition, the ways in which microcontroller operations can fail or be corrupted are discussed. The tests that will be created will take these possible faults into account.
3. *How can the ways in which the processor is affected be tested?*
   Taking into account the previously described effects, tests must be constructed to help identify these faults and where they originate. The question must be answered in a model agnostic manner, to allow different microcontroller architectures to be evaluated. The creation of methods to test microcontroller faults, is at the heart of the study.
4. *In what way do the different glitches cause different processor behaviour?*
   Once the previously created tests are implemented for a specific model of microcontroller, the results can be evaluated and used to gain insight into the faults and their origin. The goal is to be able to create a fault model that can predict the effects of glitches.

3.2 Scope

The research project is limited to four weeks, which is insufficient time to explore all the research avenues that present themselves. For this reason, the project has a very narrow scope, which is reflected in the research questions of chapter 3.1.

The project encompassed a number of fields, with which different approaches could have attempted. However, it was decided to focus the project on a number of points.

The most import goal was to create a method for evaluating effects of glitches on microcontroller operations. The method should make it possible to evaluate microcontrollers with different architectures. To help with the creation of this method, the components and their operation as well as the manner in which these can be affected by glitches, are described. These components include: memory, register, or the Arithmetic and Logic Unit (ALU). Tests were created to allow insight into the ways in which various components were affected. The tests presented are all software based. Greater insight could be gained by attaching a logic analyser to various components, which is impractical to do in most situations.

Only glitches which manifest on a short timescale are investigated. The effects of glitches are hard to predict and it is possible that these require extra time to manifest. In addition, it is expected that these will be harder to exploit.

The previous steps are the goal of this project. To underline the practical nature, the tests and method were carried out on the XMEGA64A3 microcontroller. A fault model was established, which facilitated the creation of an attack model. The attack model describes how the effects of the witnessed glitches can manipulate running software.

The glitches used for fault injection were also scoped. Common voltage glitching strategies were evaluated. These strategies were contrasted to the method used: voltage dips on the Vcc line. Voltage glitching was chosen because it can target most IC's and requires fewer resources in comparison to other fault injection methods, such as laser fault injection.

Chapter 4

Glitching methods

Fault injection is a way to induce an Integrated Circuit (IC) to function in an incorrect manner. This misbehaviour can be used to exert control over the target. A number of different methods are available when trying to induce faults in IC's. This paper will restrict itself to the discussion of the effects of voltage glitching; other methods are shown for comparison.

Methods for perturbation include: optical, voltage, thermal and magnetic.

Optical fault injection is the act of using a laser of varying wavelengths to induce faults in a target IC, an example of this is presented in [10]. The target IC must have previously been prepared for such an attack. The packaging is removed with acid, while not damaging the circuit itself. Laser fault injection can target certain areas of the IC, which localises any effect on the target.

Heat perturbation uses heat to cause faults in IC's. IC's are built with thermal tolerances in a specific range. Exceeding these tolerances causes undefined behaviour, which can be exploited. This is typically done by extreme cooling or heating. The cooling or heating of an IC affects the operation of the entire circuit and can cause one or more logical components to misbehave. An example of such glitches can be found in [11].

Magnetic fault injection is the act of inducing faults in a target IC by exposing it to a magnetic field. Depending on the field type and strength, this attack can be localised to subsystems, increasing the control that can be exerted over the target.

Finally, voltage glitching is purposefully exceeding the tolerances of voltage and timing of a target IC. The next section will discuss the various options that are available when using such an attack.

4.1 Glitch generation

This section will enumerate the different voltage glitching attacks. The research will focus on low voltage dips injected in the Vcc line of the target. During the discussion of the effects of glitches, a number of the other methods will be discussed for contrast.

Possible attack methods include: low voltage dips, high voltage spikes and long duration threshold voltage dips.

During low voltage dips the target's normal voltage level is briefly lowered, typically for a fraction of a clock cycle. A different method it to briefly raise the voltage. Another possibility is to lower the voltage of the target for a longer interval, spanning multiple to hundreds of clock cycles.

Short glitches may affect the core and the buses. A longer voltage dip might affect the writing or erasing of flash or EEPROM. Flash and EEPROM require more power to erase and rewrite than to read [12].

The discussed methods are typically applied to the following microcontroller lines. The most commonly glitched lines are: clock (CLK), reset (RST) and power line (Vcc).

A chip has a lower bound on the time it takes for its gates to stabilize during a clock. If a clock glitch is introduced, the gates are sampled before they have had a chance to stabilize, resulting in the malfunctioning of the chip. The attack is typically performed on smart cards that have an external clock line. Microcontrollers have an internal clock and can optionally connect to one. While they do support external clocks these are not always enabled. Clock glitching, while stable and effective, is not always available as an attack method.

The reset line is a way to externally reset a microcontroller. A common method is to pull the reset line low to cause a jump to the reset vector. The line must be pulled low for a specific amount of time to allow the signal to propagate through the circuit. If the reset line is pulled low for a very short amount of time, undefined effects could occur. The signal will not have had a chance to reach all the different components and these will react in unintended ways.

The Vcc line is the line which powers the IC. This paper will focus on this method of inducing faults.

The attacks above have something in common: it is extremely hard to determine what exactly is being affected. An approach to this type of attack is to set wide parameters and search these in an automated manner. Once a range of parameters that causes faults have been found, the scope is narrowed and the attack is performed again. In all of the presented attacks the temporal aspect is vitally important to control the behaviour of the target. Trying to influence the execution of code in a controlled manner requires knowledge of what is executing when a glitch is attempted.

4.2 Glitching setup

To perform the tests presented in chapter 6, a glitching test setup is required.

Figure 4.1 depicts such a voltage glitching setup. The computer records the results and controls the setup. It is connected to the voltage glitching device, in this case the VcGlitcher, an FPGA based glitch generation and control device, which is able to inject configurable glitch patterns into the controlled power line[13]. The VcGlitcher provides Vcc voltage and resets the target. The target can synchronise with the glitcher by setting a trigger and can present results to the computer using a serial interface. The use of an oscilloscope is also encouraged.

The target board must also be prepared for glitching. The attack works by temporarily lowering the voltage presented to the target. How much the voltage actually drops depends on a number of factors. One of these factors is the capacitance of the board and the target IC. Modifying the IC is rather hard, modifying the board is simpler. All the capacitors between Vcc and ground must be removed. The removal of the capacitors will make the board react more strongly to voltage variations. Figure 4.2 depicts the experimental setup.

There is a 1 µs delay between the time the target sets the trigger and the introduction of the glitch.

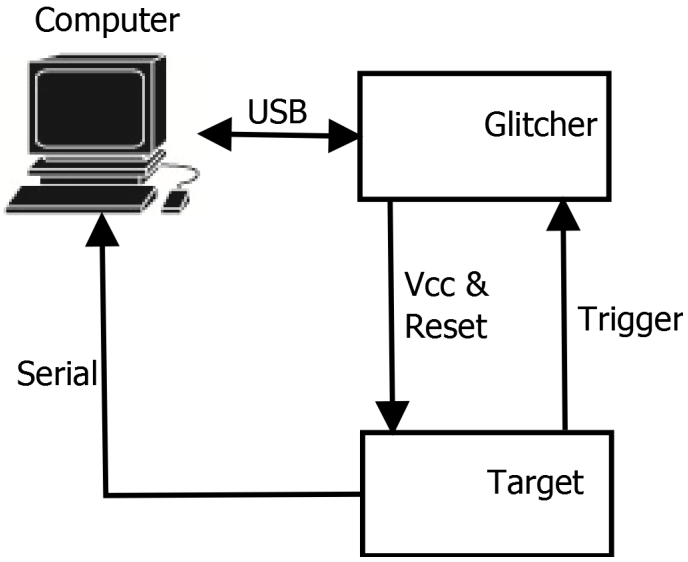

Figure 4.1: Schematic overview of the test setup

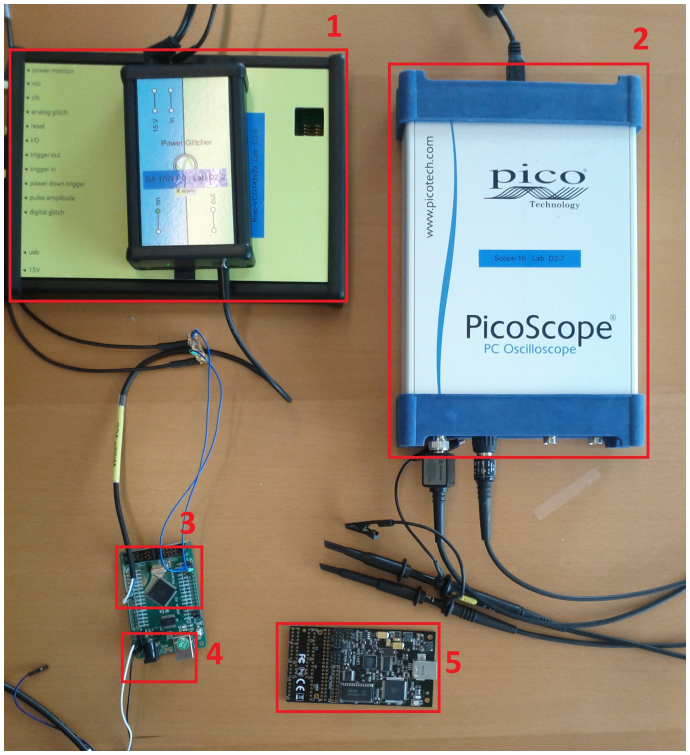

Figure 4.2: The test setup. 1: Voltage glitcher 2: Oscilloscope 3: Target IC 4: Serial connection 5: JTAG programmer

Chapter 5

Expected effects of faults

This chapter will discuss the theory behind the logical components of the MSP430 and the XMEGA microcontrollers and how their operation can be disrupted. Details of the MSP430 will be discussed followed by those of the XMEGA.

Pinpointing where faults have occurred, based on the state of a microcontroller, is extremely difficult. Compound faults can take place and are even harder to dissect. The following chapters will discuss how instrumentation can be built, which allows insight into various failures.

5.1 MSP430 operation

The members of the MSP430 family are all extremely low power embedded microcontrollers. In fact, during testing it was found that MSP430 could be powered with only the serial line synchronisation signal, which was caused by the overvoltage protection.

The MSP430 microcontrollers possess a number of operating modes that turn off peripherals to save power, some even turn off the processing core.

Microcontrollers and processors in general have a number of logical components that can be affected by glitches. This section will detail the operation of these parts and how they can be affected by glitches. Operations and logical components of a chip that could be affected by a glitch are: memory, instructions, registers and status register. A more in-depth discussion on these topics is presented below.

5.1.1 Memory

Memory can be viewed as consisting of two parts: memory itself and the memory bus. The task of the memory bus is to transport data and instructions to and from memory. The address bus selects the memory location to read from or the memory location to store to. The memory data bus transports the data stored at these locations.

The MSP430 has a 20 bit wide address bus. Thus, a total of one megabyte of address space is available. The address bus is word aligned. The data bus is 16 bits wide.

The bus is composed of the data and the address bus. Figure 5.1 shows a schematic representation. Both can fail independently of each other . If there is a fault on the data bus, incorrect data are presented. If the address bus is glitched, the data presented will be correct but from/to the wrong location. Identifying which faults occurred will be hard, especially if both occur at the same time.

When trying to separate memory faults from memory bus faults, the glitched system can be identified by the temporal aspect of the glitch.

If after glitching has stopped the data is fetched again and the correct data is fetched, the bus is at fault. However, if the same corrupted data is retrieved, the memory will most likely be at fault.

It is important to note that all instructions are fetched via the bus and stored in memory. A way to separate the bus and memory glitches from a decode glitch will have to be devised. The section 5.1.3 describes the glitches that can occur during decoding. Depending on the addressing mode, certain instructions will also send their result over the bus.

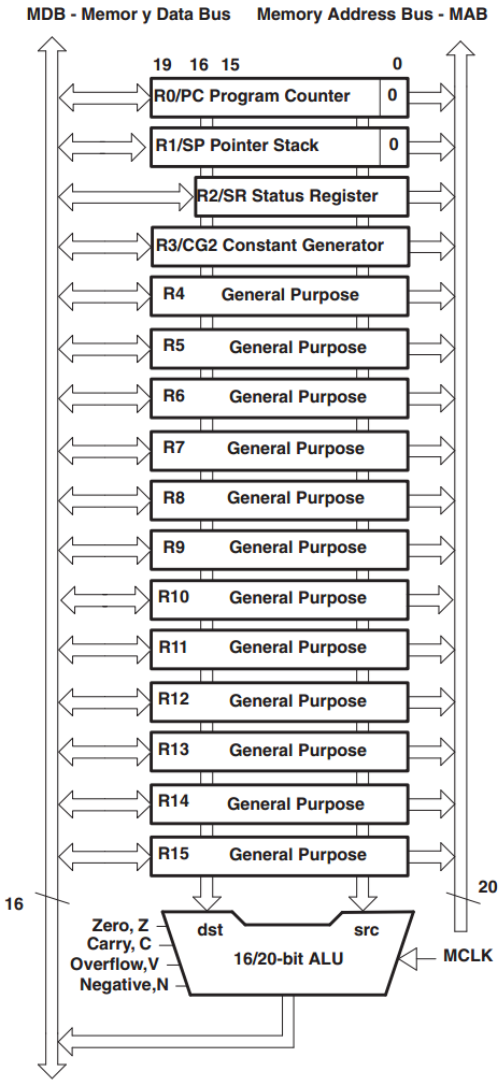

Figure 5.1: Memory address and data buses on the MSP430 source: [14]

An important distinction must be made with regard to flash memory. Erasing of updating flash commonly requires more current and/or a higher voltage [12]. The updating of flash commonly requires a block to first be erased. This erasing of data is what actually requires the extra power.

5.1.2 Registers

Registers are an important part of MSP430 execution. The MSP430 contains a number of special purpose registers as well as a number of general purpose registers. Registers can be glitched in mainly two ways: an incorrect instruction is executed or a static corruption occurs. The section 5.1.3 describes the glitches that can occur during instruction execution. However, most of the instructions have registers as operands. The MSP430 has a number of special registers which can significantly alter the behavior during glitches. These are detailed below.

General purpose registers

The General Purpose (GP) registers can be the source or destination of any instruction. They are 20 bits wide.

Program Counter

The Program Counter (PC) points to the memory location currently being executed. It is 20 bits wide and word aligned, meaning its lowest bit is always set to 0. The instructions CALL and RET can be used to call and return from functions. In addition, the instructions that operate on the GP registers can be used.

Setting the destination register to the PC with any normal instruction will effect a (computed) jump. This will be particularly difficult to diagnose and will probably present itself as a device reset.

Stack Pointer

The 20 bit Stack Pointer (SP) points to the stack. As with the PC, the SP is word aligned, meaning that it can only point to even addresses. Hence, any data stored on stack must be a word multiple. The `CALL` and `RET` instructions modify its value. In addition to these special instructions, the SP can be used as a source and destination of the normal instructions.

When instrumentation is designed, it will be important to keep in mind that a corrupted SP will cause the stack to point elsewhere. Adding data to the stack should be fine. However, data retrieved from the stack after glitching could be from an entirely different location. It is therefore preferable not to assume anything about the state of the stack when making instrumentation.

5.1.3 Instruction execution

Instructions are commonly categorised into three groups: flow control, memory operations and Arithmetic Logic Unit (ALU) operations. Flow control operations include instructions like branch and jump. Memory operations either store or load data from memory and ALU operations are computational operations. The MSP430 is not a strictly RISC architecture therefore ALU instructions can be performed on memory locations. The possible problems for both categories will thus apply.

All instructions independent of their category are executed according to the Von Neumann cycle. The Von Neumann cycle divides the execution of an instruction into a number of phases: fetch, decode, execute and store.

The fetch phase of the instruction execution cycle retrieves instructions from memory. This subject is discussed in greater depth in section 5.1.1.

The phase of the instruction execution cycle, which decodes the opcode and routes the operands to the correct hardware for execution, is called the decode phase. The decode phase assigns meaning to the data retrieved from memory. If this memory is incorrectly fetched or decoded, the result will be the execution of a different or illegal instruction. Some processors have ways to catch the decoding of an illegal instruction. The separation of faults in the fetch and decode phases can be difficult to unravel. Decoding the address mode incorrectly could result in a single word instruction being decoded as a double word instruction or vice-versa.

For clarity, an encoding of an instruction is presented. The first chunk is the opcode. The opcode describes what operation will be performed and it is followed by the operands. In this example, the register operand is encoded as a nibble. The register being written to or read from is thus easily recognisable in the hexadecimal representation of an instruction. This is illustrated in figure 5.2. Here, the nibble '6' represents the register R6 and the nibble 'E' represents the register R14; figure 5.3 shows the registers and their mnemonics.

During the execute phase of instruction execution, the processor performs the operation which it has decoded.

Faults introduced during the execution phase would manifest themselves by computing an incorrect result. A distinction must be made between an incorrectly decoded or fetched instruction and an incorrectly calculated result. A decoding failure results in a different instruction being executed, as described in section 5.1.3, perhaps with different operands. An incorrectly calculated result would not have a strong relationship with the original instruction.

The store result phase of the instruction execution cycle stores the result from the executed instruction in a register or memory. A glitch during this phase would manifest itself as a corrupted or un-updated value. Memory glitches are discussed in greater detail in section 5.1.1.

In practice, the distinction between fetch and decode faults will be hard to make. Using timing information about when the glitch was injected, will allow some untangling of the faults.

```
464E                MOV.B   R6,R14
```

Figure 5.2: Move instruction and its binary presentation

Number	Description	Mnemonic
0	Program Counter	PC
1	Stack Pointer	SP
2	Status Register	SR
4-15	General purpose registers	

Figure 5.3: Registers of the MSP430 and their mnemonics

5.2 Target switch

Initially, the MSP430 was chosen as a target for the attacks. However, the MSP430 was not susceptible to voltage glitching. The section 9 details the reasons why. The goal of the project is to create a fault model not to create glitches. With this in mind it was decided to switch targets. The new target is the XMEGA64A3 from Atmel. The operation of this microcontroller is detailed in the next section.

5.3 XMEGA operation

This section details the operation of the XMEGA microcontroller and the ways in which these can fail. Because there are considerable similarities between the possible faults of the XMEGA and the MSP430, only the differences will be discussed. The XMEGA is an 8-bit microcontroller with a 16-bit address bus.

5.3.1 Memory

The XMEGA line of microcontrollers and its predecessors, the MEGA line, use a modified-Harvard-architecture [15]. This means that there are multiple memory spaces: program memory and data memory. The program memory is flash based and contains the instructions and their immediate data. The data memory is SRAM based.

The 'modified' in modified-Harvard-architecture refers to the fact that it is possible to load data from the instruction memory. This requires the use of special instructions.

The XMEGA has a 16 bit program address bus and an 8 bit data bus. The program memory has 16-bit data and address buses.

The memory faults that were detailed for the MSP430 in section 5.1.1 are also applicable to the XMEGA.

5.3.2 Instructions

Most of the theory regarding instruction execution on the MSP430 is also relevant here. There are a number of crucial differences.

The XMEGA line of microcontrollers conforms to the RISC architecture [16], which among other things, means that only store and load instructions operate on memory. This means that all data manipulation operations must be performed in registers. For example, if the variable x is to be XOR'ed with the contents of variable y the following instructions will have to be executed:

1. lds r12,x
2. lds r13,y
3. xor r13,r12
4. sts r12,x

All instructions take at least one cycle to complete. An instruction consisting of two words takes an extra cycle to complete. An instruction that stores or retrieves from SRAM takes an additional cycle to complete. An instruction that causes a pipeline-stall by changing the PC will take an additional cycle. The call and return instructions are exceptions to these rules.

The XMEGA uses a two stage pipeline. This means that while one instruction is being executed the next one is already being fetched. This feature significantly improves the performance of the XMEGA.

It is important to note that the XMEGA does not have an illegal instruction exception handler. Any illegal instructions encountered will be treated as a `NOP` [8].

5.3.3 Registers

The XMEGA contains 32 general purpose registers. These are all 8-bit wide. Some registers have special purposes.

The upper 16 registers can be computed on with immediate values. The upper four register pairs can perform word addition with immediate 6-bit operands. The upper three register pairs are called the index registers and can be used to address data or flash memory. The result of any the multiplication operations are stored in the R0:R1 register pair.

Unlike the MSP430 the special purpose registers SR, PC and SP are mapped into IO memory.

Chapter 6

Detecting effects of attacks

This chapter describes how to create tests and instrumentation. The test is the instruction or instruction sequence, which will be evaluated. The instrumentation is the code that will output the resulting state of that sequence. Care must be taken not to accidentally glitch the instrumentation. Before the instrumentation is discussed, an overview of the possible data sources is given. With these sources in mind, instrumentation is constructed. Finally, a number of different tests are presented. This chapter uses the theory of operation and possible faults from the previous chapter. The results of the presented tests can be found in the next chapter.

Section 5 discussed the possible effects of glitches on a logical component level. However, conclusively attributing faults to different components is impossible. The effects seen here will be modeled in different domains. Effects will be defined in terms of the following domains: memory, instruction execution, registers and flags.

6.1 Where can we detect faults?

Faults can cause different logical components of microcontrollers to function incorrectly. This paper restricts itself to the effects of faults as can be seen from software. Faults can only be detected after they have transpired; these faults must therefore be inferred from the resulting state.

The main regions of interest with regard to detecting faults are:

1. Main memory
2. Program Counter (PC)
3. Stack Pointer (SP)
4. General purpose registers
5. Special purpose registers (Status Register/Status flags)

The instrumentation and tests of this section will aim to capture as much of the state as is relevant from these locations.

6.2 Instrumentation

This section will describe the way in which instrumentation can be constructed and what measures can be taken to avoid glitching the instrumentation. Sometimes, glitching the instrumentation cannot be avoided, in such cases it must be detectable.

The general structure of the instrumentation will be as follows:

1. Initialisation of peripherals;
2. communicating a constant;

3. initialisation of variables;
4. setting the trigger;
5. the test;
6. clearing the trigger;
7. sending the gathered data;
8. sending a constant.

Initialisation of peripherals is specific to the model of microcontroller being targeted. At the very least a communication channel such as a UART and a IO port must be initialised. The IO port must be configured so that it can set an output pin which will act as a trigger.

The UART is used to communicate the results of the test. An extremely simple communication protocol is employed and its rationale is discussed in section 6.2.1. Initialisation of peripherals can be performed in the C language.

Variables used during the tests must also be initialised. Apart from the initialisation of variables in RAM, register values must also be initialised. These can best be set to unique and easily identifiable values. The precise values depend on the test being performed. Memory values can easily be set from C, while register values must be initialised in assembly.

After this the trigger is set. The trigger signals that the critical section will now be executed. The trigger will act as a synchronisation point for timing related parameters. It is vital that the time between the setting of the trigger and the start of the test, is known, is as short as possible and contains as little jitter as possible. The setting and clearing of the trigger can best be done from assembly. The time between the trigger being, set and being reset is the critical section.

The critical section is composed of the instructions that are targeted by the glitch. To have as much control as possible over these instructions these must be written in assembly. After the critical section the trigger is reset.

Next are a number of `NOP` instructions. These are to ensure that the instrumentation is not accidentally glitched. It is hard to discern between glitched tests and glitched instrumentation. It also gives the processor time to restore its power level.

Next, the gathered data is sent. It is impossible to send the exact state of the microcontroller after the glitch. The following values will be lost to some degree: PC and memory. While the value of the PC can be calculated, memory must be sacrificed so that the registers can be safely sent. In practice, all the GP registers, the status flags and the SP are pushed onto the stack and then sent. There are two methods which can identify corrupted results: send the data twice and run the test twice. In the first case, the data is sent twice and compared against itself. In the second case, the values are sent only once but, the test is performed twice. The first test is performed without glitching to get a baseline reading. All subsequent tests can be compared to this baseline. The second method is preferred because it requires less maintenance.

The delay between the setting of the trigger and the earliest possible glitch must be measured. If this delay is multiple clock cycles, an equal number of `NOP`s must be inserted to ensure that it is possible to target the test instructions.

If the target microcontroller has interrupts, these can be disabled and programmed to send a recognisable value. Because the interrupts are disabled, any interrupt firing will be caused by a glitch.

If the microcontroller supports exceptions that allow undefined/illegal instructions to be caught, these must be enabled.

6.2.1 Communication

To analyse the state of the processor, data from various locations must be extracted in an automated manner. This will commonly be sent over a serial interface such as the UART. An extremely simple communication protocol is employed:

- Constant preamble;
- data;
- constant ending.

Because the target electronic circuit is operated outside of its specifications, the IC will often react by resetting itself. Such resets will occur and must be detected. The constant preamble is to make detection of a reset straightforward. Next is the data that is to be analysed. After this, a constant ending is sent. The task of this constant ending, is to be easily detectable. A corrupted constant indicates that a fault has occurred outside of the glitch window. This can be caused by the microcontroller's UART running out of power. If corruption of the constant is detected it casts significant doubt onto the integrity of the previously received values. This paper will discard results which have a corrupted constant. Results which contain two preambles and only one set of data values will be considered resets and will not be investigate further.

It is prudent to ensure that all the data has been successfully sent before setting the trigger. This will ensure that the glitch does not interfere with the data sent. It also allows an interrupt to commence sending a single byte immediately, which will help its detection.

6.2.2 Reaction delay

Timing is a very important factor in glitching. It is therefore important to know what the timing characteristics are of the testing setup and the instrumentation. It is imperative that the time between the trigger being set and the glitch being delivered are known. The timing data will be used to ascertain during which instruction the glitch was introduced.

6.3 Tests

This section will describe the tests that can be performed and what can be inferred from them. This section aims to be target agnostic. The tests that were performed on the XMEGA can be found in chapter 7. First, tests that can narrow down the timing window, are presented. Once, the timing window has been narrowed, it is possible to test of individual instructions.

6.3.1 Improving the glitch

The timing window and other parameters to glitches need to be established. The tests presented here allow these to be performed. The tests are primarily aimed at establishing relationships between the introduction of a glitch and its effect.

Simple add sled

Before analysis of faults can commence, it must first be proven that glitches can occur. Furthermore, the robustness and repeatability of these glitches must be established. This simple test will detect if any faults occur. This can be used to create a model of the types of faults which occur in the target using the specified attack. The name 'add sled' is derived from the name NOP-sled presented in [17]. The name is not truly deserved at this stage. Later on, it will be used in conjunction with branch and jump instructions and it will be more deserved.

The 'add sled' is nothing more than an unrolled increment loop. A register is chosen as a counter and repeatedly incremented by a constant value. In practice, this means a number of `ADD` or `INC` instructions consecutively executed. The precise instructions that must be executed depend on the instruction set of the target.

The test has purposefully been constructed in a manner that makes it impossible to tell which instruction was successfully glitched. The following cases can be detected:

1. Instruction skipping;
2. related register corruption;
3. unrelated register corruption;
4. status flag corruption.

The starting value of the register, as well as the number of add instructions and their operands allow the precalculation of the result. If the value is not equal to this result, a successful glitch has occurred.

For instance, if a counter is initialized at 0xF0 and is incremented five times, the value 0xF5 should be output. However, if the value 0xF4 is repeatably and consistently output, it can be assumed that an instruction was not correctly executed.

Using the previous example, if the value 0x00 is received, it can be concluded that the register's content was corrupted. This is assumed because there does not appear to be a relation between the expected value and the instructions which should have been executed. This test does not give an indication as to why the register's content was corrupted. The tested instructions could have been incorrectly fetched, decoded or executed. Further tests should be conducted.

A simple variation is to use a different register as the counter. The IC can have different electrical properties for different registers. This can be explained by the possibly different functions registers can have. This allows one to more easily classify a result as a related register corruption or unrelated register corruption.

If during the previous tests, other registers were affected, this will be referred to as an unrelated register corruption. An unrelated register corruption bears no direct relation to the instruction being executed. In other words, it is neither the source nor the destination operand.

An interesting variant of this test is only a NOP-sled. Any register variation will be caused by register corruption. This does not rule out instruction decoding, fetch or execution faults.

Status flag corruption is unlikely to be detected; this is due to the fact that `ADD` and `INC` instructions commonly set or reset these flags, depending on return value.

Or sled

Once a repeatable and robust glitch has been found, the timing relationship of the glitch with the executed code must be improved. A successful glitch in the 'add sled' test does not tell us which add instruction was glitched. For instance, using the previous example of a starting value of 0xF0 and five increment instructions, a result of 0xF4 is computed. This does not indicate which of the five instructions was not correctly executed.

The 'add sled' test should have established in what manner simple instructions are likely to be glitched, as well as which registers can become corrupted. The timing relationship of the glitch with the executed code must now be improved upon, so that it is possible to pinpoint the precise instruction that is affected. This will allow precise targeting in consequent tests.

For this reason, the 'or sled' has been devised. The 'or sled' is the repeated execution of `OR` instructions with different immediate operands. A register is set to zero and repeatedly or'ed with consecutive powers of two. At the end of the instruction sequence, all the bits in the byte or word will be set. If one of the bits is not set, it indicates that the corresponding instruction was glitched. More than a couple of glitched instructions could indicate that register corruption occurred. The timing information deduced in such a manner will be of great use in further tests. In particular, the isolation of the attack to a single instruction will allow the number of instructions in the critical section to be decreased, making it possible to read the status register before any other ALU operations are performed.

A number of variations are proposed that can be applied if the target architecture does not support `OR` instructions with immediate operands or immediate operands of sufficient size.

If there are only 8-bit `OR` instructions available as is common in 8-bit architectures, a sequence of only eight instructions can be constructed. An extra register can simulate the use of a 16-bit register, allowing a sequence of up to 16 instructions to be constructed, which sufficient for most cases. When selecting the registers it is prudent to avoid the registers that were corrupted in the previous tests.

If immediate values cannot be used, other registers can be set up to contain the values needed. While it is uncommon to not have an immediate `OR` instruction, the following variation uses `XOR`. The `XOR` instruction does not always have an immediate variant available. Again care must be taken when selecting registers. Registers that are prone to corruption are best avoided.

6.3.2 Instruction groups

When enough control has been established over the glitch that single instructions can be targeted, it becomes possible to characterise the effects. This section will try to define test scenarios to identify the different effects that could take place.

The list presented in section 6.3.1 describes the possible effects that can occur. It is reprinted here for clarity.

1. Instruction skipping;
2. related register corruption;
3. unrelated register corruption;
4. status flag corruption.

This section will define a way to gain greater control over the characterisation of related register and status flag corruption . Ways in which the corruption of unrelated registers occurs will also be investigated.

Related register corruption can be caused by the following: incorrect instruction decoding/fetching, incorrect decoding/fetching of source or destination operand or an execution fault.

Incorrect instruction decoding and fetching describe the possibility that the correct operands were used and the correct result was computed, only for a different instruction. For example, the instruction for multiplying 3 and 4 might be transformed into the instruction to add 3 and 4. The answer would then be 7 instead of 12.

The incorrect decoding of source or destination operands refers to situations where the correct instruction has been executed but with incorrect operands, specifically the use of incorrect source register operands.

An execution fault refers to a correct instruction, executing with the correct source and destination operands calculating an incorrect result. Ideally, the goal should be to predict the corruption of the result.

It is important to note that a lot of these effects cannot be easily attributed to one cause or the other, nor will it possible to prove conclusively that an effect occurred during execution or fetching. In particular, when multiple effects occur it will be extremely difficult to correctly identify and characterise these effects. This will make it exceptionally difficult to predict what the effect of a successful glitch will be. When effects cannot be accurately predicted, it will be referred to as corruption.

Arithmetic and logic instructions

Multiply instructions The multiply instruction is a good test case for demonstrating the methods for identifying what effects have taken place when a corrupted result is encountered.

If the destination operands or registers are corrupted, it might be beneficial to identify if this is caused by the use of incorrect source registers.

To verify that this is the case, all the registers are filled with unique prime numbers and the instruction is executed and glitched. If the result of the corrupted operation is a composite number, that is the product of two primes, it can be assumed that the operation used different source registers. The chosen prime numbers are best chosen as large as possible to decrease the number of false positives.

Flow control instructions

The flow control instructions are the set of all instructions that influence the program counter beyond merely incrementing it. This includes all branches, skip and procedure instructions.

A number of cases are expected: the instruction is not correctly executed, the operand of the (near) jump is corrupted or the PC is corrupted.

If the instruction is not correctly executed, it can be identified using the techniques discussed in the previous sections. In the case that the PC is corrupted, it will be quite hard to identify in what manner it has been corrupted, because the PC could conceivably point anywhere. A true corruption will be hard to translate into an attack. If the operand of a near branch or jump is corrupted, it will be detectable and potentially exploitable.

This can be done by building an 'add sled' as presented in previous sections. Now the add sled must be placed before and after the target instruction. For instance if the near jump has a range of -63 to +64, 63 instructions must be placed before and 64 after it. To verify if a corrupted branch jumped backwards or forwards, it is advised that the preceding block increment one set of registers and the following block another.

In order to prevent the clock skew of the target and attacker becoming too great, it is possible to jump directly to the target instruction. Jumping directly to the target instruction, must only be attempted if the glitch is precise enough not to glitch the first jump. Alternatively, it is possible to jump to just before the target instruction. The extra executed instructions will have to be accounted for.

Memory instructions

Instructions that access memory are an important class. Influencing the way microcontrollers load and store data from memory can give significant control over applications running on them. Typical instructions in this category are: `PUSH`, `POP`, load and store.

To which degree memory instruction faults can be categorised, depends on the target architecture. There are a number of architectural concepts which can increase the insight into memory related faults. These are: the Memory Management Unit(MMU), Memory related exceptions and different address spaces.

Larger processors often contain a MMU. If a process tries to read memory belonging to another process, the MMU will prevent this and issue an exception. This will impact any tests that affect the address which is written to or read from. Issuing exceptions or interrupts for illegal memory access, can also occur on processors that do not have an MMU. An example of such an exception is trying to write flash memory in an incorrect manner. If such exceptions can be caught by the test program they can yield significant insight.

It is not uncommon to have multiple memory spaces: IO, data and program. These memory spaces will have separate instructions associated with them. In other cases, these different memories will be mapped into the same address space. A glitched load instruction could read memory from the instruction or IO space.

Also of importance is the distinction between load-and-store architectures and register-memory architectures. Load-and-store architectures require data in registers before a computation. Register-memory architectures have instructions that can compute using data from registers or memory. Typically these instructions have a number of addressing modes. The following tests will only discuss load and store for simplicity. Glitches introduced in more advanced instructions can be hard to untangle, it is therefore recommended to first evaluate the memory mechanisms themselves.

Load In general it is hard to distinguish between reading from an incorrect location and bus corruption; the presented test will try to give some insight. Further tests will have to be designed on a case by case basis.

A variable called the target is initialised in memory. It is preceded and followed by large blocks of memory with distinct values; referred to as the background. The preceding background block and the following background block are given different values. The goal is to make these blocks as large as possible while still allowing the program to function. Care must be taken to leave enough room for the stack and other variables. The memory layout declarations can be written in C. Verifying the output of the compiler is recommended.

During the critical section the target is loaded into a register using a move or load instruction.

If the register contains values used in the background blocks, it is proved that the instruction has read from a different location. If only values from the lower block are seen, it could be conjectured that bits are being cleared. If only values from the following block are seen, it can be conjectured that bits are being set. If values from both blocks are seen the read address can only be considered corrupted.

If the read value is not from the background blocks, the root cause will be hard to track down. A number of possibilities present themselves: the retrieved value is corrupted, the value is read from the memory that has not been initialized, or the value is read from part of the address space that is not backed by memory.

Tracking down the cause of a corruption is not always possible. The value could have been read from a memory location that has not been initialized, for instance the stack or data sections. It is also a very real possibility that a value has been read from a location that is not backed by memory. If the target architecture maps multiple memories into a single address space the read could have been from a different memory. For example, a read from SRAM was attempted and IO memory is read.

Store The test presented in this section, will try to identify how a store instruction is glitched. Most of the set-up presented in the previous section can be reused.

All of the available memory on the device is initialised to known values; this is called the background. A variable, called the sentinel, is placed in the middle of the background and initialized to a different value. During the critical section, a value is written to the sentinel. This value will be referred to as the target. Care must be taken to ensure that the target is not present in memory before the test starts.

After the critical section, the program searches through the background for a changed value and the sentinel is also read back. If the target is found in the expected location, no glitch occurred. If the target is found at a different location, an address bus fault or instruction fetching/decoding fault occurred. If the target is not found, a number of possibilities present themselves. The sentinel was corrupted; indicating a data bus corruption. Another possibility is that the instruction was not correctly executed. The available power could have been insufficient to correctly update

the value in memory. The address to which the write was redirected, may have been outside of memory. Depending on the architecture the value could have been written to the IO space or part of the address space not backed by memory. Finally, both a data and address bus error could have taken place. If the address glitch was within the background, it is possible to locate the corrupted value.

If the target architecture is not Harvard-architecture a memory search will also search through the program code. It must be ensured, that the target value is not literally present in the code. Using a simple computation will be sufficient to prevent this. Care must be taken to ensure no memory is written before the memory has been searched. Stack memory commonly grows down; it is therefore advised to search from the lowest memory to the highest memory address. This will minimise the effects of values on the stack; in particular return addresses can change when different variations are tested.

Chapter 7

Effects of attacks on processor behavior

This chapter details the results and their analysis of the tests presented in chapter 6. The previously discussed tests were designed in a microcontroller agnostic manner. The presented results were obtained on specific microcontroller architectures, families and models. Due to architecture or model limitations it is not possible to fully implement the previously given advice. Any limitations and their workarounds will be given.

The results obtained will be the basis for the attack model presented in the next chapter.

Instruction execution is normally divided into fetch, decode and execute phases. The granularity of the instrumentation is at best a single instruction.

The Atmel instruction set guide [18] can be referenced for information pertaining to instruction execution. These tests were conducted on the experimental setup presented in section 4.2.

The current hardware setup the length of a glitch cycle is 1 microsecond. The XMEGA runs at 2 MHz, which translates into two instruction cycles per microsecond.

7.1 Glitch profile

Before the logical effects of a glitch can be investigated the glitch profile of the target must be established.

Figure 7.1 shows the glitch profile of the XMEGA64A3. A number of glitches are introduced and in reaction the Vcc voltage drops. The voltage always recovers before the next glitch. Meaning that the each glitch cycle is independent of the previous one. On microcontrollers where the glitches are not independent of one another, each successive glitch lowers the voltage further. At some point the voltage is lowered to a level where a fault is introduced.

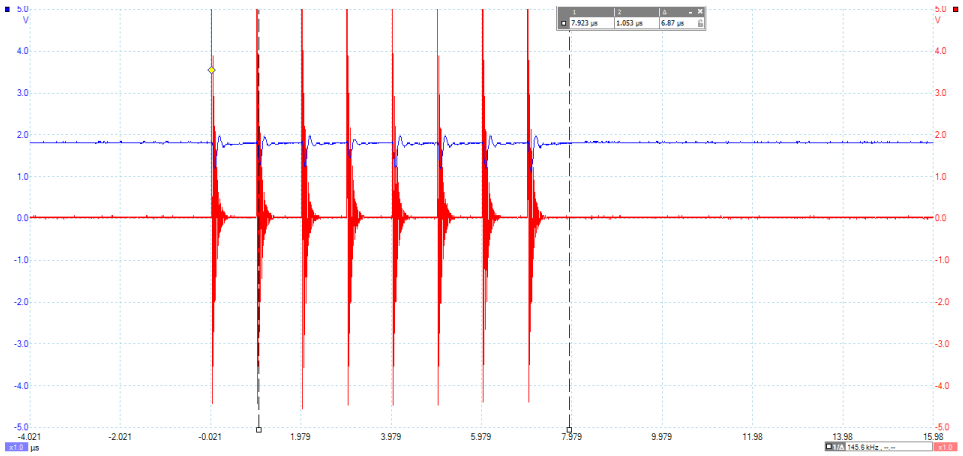

Figure 7.1: The glitch profile of the XMEGA64A3. The red line represents the digital glitch and the blue line represents the Vcc voltage.

A number of parameters must be determined: glitch length, glitch offset, glitch delay, Vcc voltage, glitch voltage and glitch cycles.

Figure 7.2 depicts the timing of glitch parameters. The parameters are explained below. The glitch voltage is the amount by which the voltage will be modified during a glitch. The glitch length is the amount of time to apply the glitch voltage. The glitch offset is the offset within a 1 µs division after which the glitch must be applied. The glitch delay is the amount of µs to wait before administering the glitch. The XMEGA runs at 2MHz, which makes each clock 0.5 us. The Vcc voltage is the voltage used to power the IC. The glitch cycles defines the number of times to repeat the glitch.

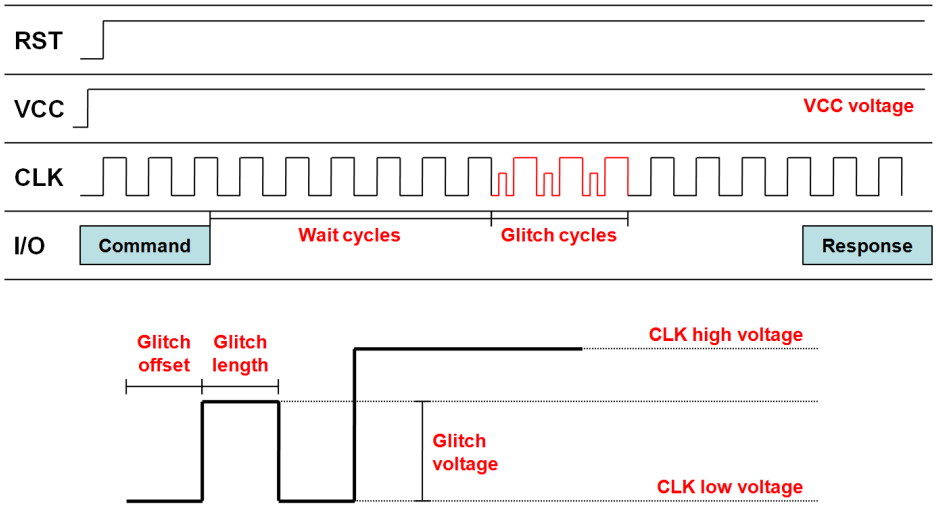

Figure 7.2: Glitch parameter timing

7.1.1 Simple add sled

Here are the results from the add sled presented in 6.3.1. Is a test to find out what the best parameters with which to generate glitches.

Figure 7.3 is correct output of the test program. This has been collected before testing began. It shows Stack Pointer High byte (SPH), Stack Pointer Low byte (SPL), the Status Register (SR) and the general purpose registers R0-R31. Figure 12 shows some of the different results that can be obtained by glitching the 'add sled' program from section 6.3.1.

The R30 and R31 contain the counter. A changed value indicates a successfully glitched program. The test results were varied and indicate that a number of different cases occurred.

1. A single ADD instruction was glitched
2. A multiple ADD instruction was glitched
3. An unrelated register was corrupted
4. The status register was corrupted

7.1.2 NOP sled

To investigate the register corruption a similar test was conducted. The goal of this test is to separate the effect of the not correctly executed instruction from the register corruption. This will hopefully give credence to the assumption

```
SPH-2F SPL-D2 SR-00  R00-FF R01-00 R02-00 R03-00 R04-00 R05-00
R06-00 R07-00 R08-00 R09-00 R10-00 R11-00 R12-00 R13-00 R14-00
R15-00 R16-00 R17-20 R18-08 R19-00 R29-02 R21-00 R22-02 R23-5E
R24-60 R25-00 R26-07 R27-00 R28-04 R29-20 R30-F6 R31-00
```

Figure 7.3: Correct output of the 'add sled' program.

that the effects are unrelated to each other. It will hopefully also separate the register corrupted from the instruction being perturbed. This test will allow characterisation of register corruption. This will allow the separation of instruction corruption from the noise of register corruption.

For this test a number of `NOP` instructions were placed into the test section and glitches were applied. The `NOP` instruction does not operate on any register, any corruption is thus not caused by the incorrect execution of the instruction.

In the XMEGA instruction set the opcode for `NOP` consists entirely of zeros [18]. The XMEGA's `NOP` is thus a real no-operation and not an instruction that has no effect. In contrast the MSP430's `NOP` is `MOV R3,R3`. Glitches which would pull the opcode down to zero would thus have no effect.

Glitching during the `NOP` instructions proved possible; register contents were corrupted. This indicates that the registers do not have the required power to operate correctly.

7.1.3 Or sled

The previous test only proves that glitches can be created. It yields no insight into the timing relationship between the glitch and its effects. To identify these characteristics, the test previously described in section 6.3.1 was employed. This test utilises the results from test 7.1.

The XMEGA microcontroller family does not have a 16-bit 'or' operation. Therefore, the tested version employs 8-bit immediate or's over two registers.

It was decided to use a single glitch cycle to help pinpoint the place in time where the resulting fault is introduced.

It will also be hard to say with certainty which glitch caused which effect. Figure 12.3 shows the obtained results. It is important to note that during this test, register corruption was detected. This does not have any bearing on this test, which was conducted strictly to identify timing of the glitch.

Figure 12.3 shows the output of a number of successful glitches. Registers R30 and R31 were the destination registers during the `ORI` operations. Because the bits were set from LSB to MSB, the glitched instructions can easily be found. There are three separate groups of glitches:

1. Starting at 0 ns, bit 1 is not set
2. Starting at 432 ns, bit 2 is not set
3. Starting at 924 ns, bit 3 is not set

These results all have a single bit error in register R30 and can be grouped in terms of timing and result. It is immediately clear that we are looking at glitches in separate instruction cycles. It is important to note that bit 0 is never glitched.

The window for the start of group 1 starts at zero ns. It is believed that this should be extended into the negative.

Using these results, it is possible to build a model for the timing of glitches. Figure 7.5 depicts the insertion of a successful glitch and its effect. The effect is not seen on the currently executing instruction but rather on the next instruction. Figure 7.4 depicts the timing of the fetch and execute phases of XMEGA instructions. Results indicate that the fetch phase of the instruction is glitched; more evidence in support of this is presented in section 7.2.1.

It is now possible to build a test in which single instructions can be tested, and side effects evaluated; this includes the status register.

7.1.4 And sled

To validate results, an alternate test was devised. Instead of starting at zero and or'ing with a power of two, an initial value of 0xFF is selected and repeatedly bitwise-and with the power of twos inverted. This means all values have all but one of their bits set to 1.

Using the timing parameters gained in the previous step, the third `ANDI` instruction was chosen as the target instruction.

When glitches were detected, the value was consistently as expected, a high value in bit three, giving a value of 0x4. Targeting the fourth `ANDI` instruction resulted in the expected value. This indicates that the parameters found during the previous test, can be employed in further tests.

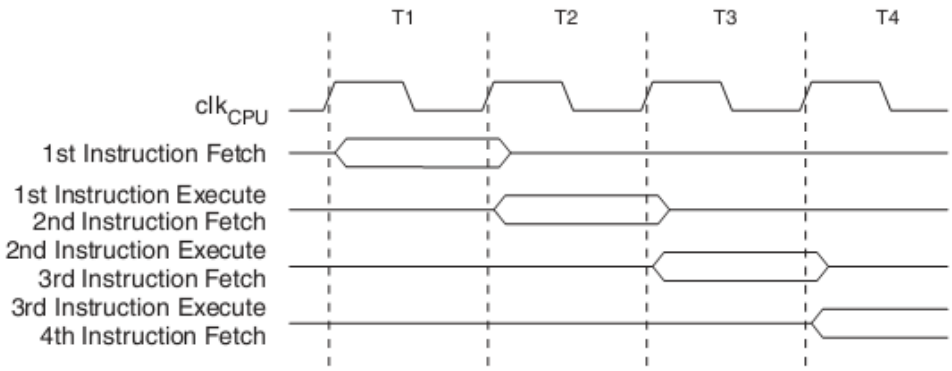

Figure 7.4: The fetch and execute phases of the XMEGA *Source: Atmel XMEGA A Microcontroller*

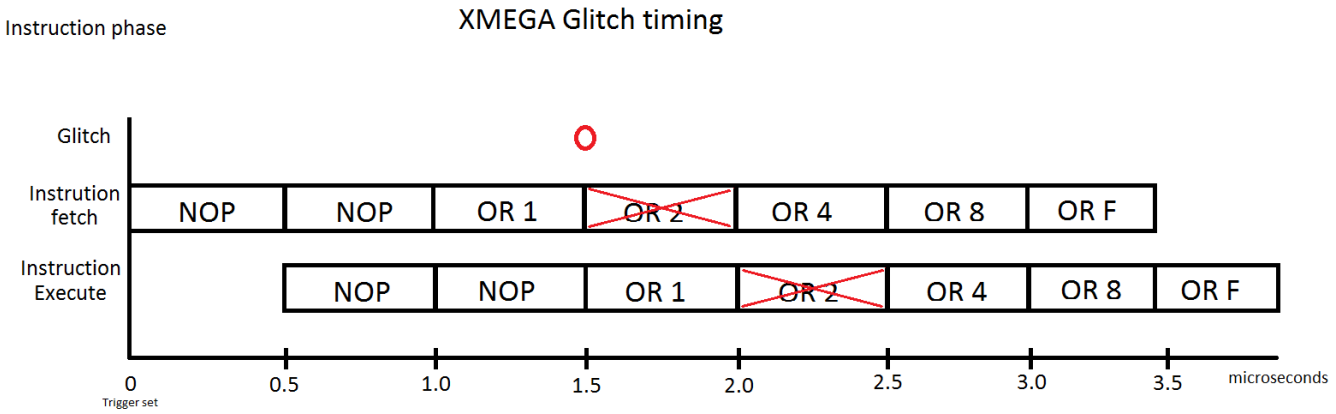

Figure 7.5: Timing model of a successful glitch

7.1.5 Exclusive or

The exclusive or operator does not allow the use of immediate operands. Setting up registers to use for this purpose gives the predicted result.

7.2 Single instructions

The previous sections show how the timing relationship between the electrical glitch and its effect are established. That knowledge allows the targeting of individual instructions presented in this section. The presented results will be used to create the attack model in chapter 8.

7.2.1 ALU instructions

ALU instructions are the instructions that require the ALU to complete. This includes all instructions that carry out a computation. Common instructions in this group are `ADD`, `XOR`, `OR`.

Multiply

In the XMEGA instruction set, the `MUL` instruction multiplies to unsigned 8bit integers with each other and places the result in R0 and R1. The high byte is placed in R1 and the low byte in R0.

The `MUL` instruction is one word long and takes two clock cycles to complete. The timing of the glitch indicates that the fault can only be successfully introduced in the fetch phase of the instruction execution cycle and not in either of the execution cycles. The glitches manifest themselves as corruption of the result registers R0,R1 and the status register. We make the hypothesis that the fetching of the instructions is affected in such a way that different source

registers are used. In other words, the output is correct, the execution is correct but the inputs were wrongly chosen. The experiment in the next section will attempt to verify this hypothesis.

To identify which registers were chosen, all registers were filled with unique prime numbers and the multiply instruction was glitched. If the result in R0:R1 was a composite number with only two prime factors, the source registers could be identified. Note that in at least one case a single operand is multiplied with itself. That means both original source operands were corrupted into the same value. It is hard to say with certainty which of the operands is transformed into which resulting operand. To tackle this ambiguity, the smallest hamming distance [19] is considered to be the valid one. In the case that there are more possibilities with the same hamming distance, only one will be chosen. The evaluation of the `ADD` instruction does not have this ambiguity.

A number of output states are shown in appendix 7.2.1.

Figure 12 in the appendix shows the output when a `MUL` is glitched. Note that in no case is there a need to set a zero in the original operand to a one in the resultant operand. This indicated that there is a tendency for one bits in the instruction to go to zero when glitched.

Add

To further verify the hypothesis presented in the previous section, a variant of the experiment is performed, targeting the `ADD` instruction. The hypothesis is falsifiable if the corrupted values are not the result of an addition with a register.

The test is setup similar to the previous one. This time, the registers are loaded with unique values. A changed value in a register indicates that it is the destination operand. Subtracting the original value of the register, gives the value of the source register. The carry flag should be taken into account if an overflow occurs.

Figure 12 in the Appendix details some of the resulting ending states of the glitched tests and the instructions that must have been executed. The results indicate, the hypothesis is valid, i.e. values found in corrupted registers can be adequately explained by the use of different registers. As an interesting side note there was never an occurrence where a higher register was chosen. This indicates a tendency for one bits to corrupt to zero bits.

7.2.2 Flow control instructions

The glitching of the flow control instructions can be evaluated using the methods described in section 6.3.2.

The ATXMEGA family of microcontrollers has branch instructions, skip instructions and procedure instructions.

The branch instructions are instructions that check one or more flags in `SREG` and conditionally jump between -63 and +64 words. The `JMP` instruction, presented in the next section, is unconditional and can jump between -2K and +2K words.

The skip instructions conditionally execute the instruction directly after it. This allows for efficient coding of small if-else branches. The combination of a skip instruction with the `RJMP` instruction emulates a far jump. The *avr-gcc* compiler generates such code sequences.

Relative jump

The XMEGA's `RJMP` instruction is an unconditional relative jump. It can jump between -2K to +2K words. Avr-gcc generates code that combines the `RJMP` and `JMP` instructions with the skip instructions (see section: 7.2.2 to simulate conditional far jumps).

The relative jump or `RJMP` instruction will consistently not be executed if a glitch is introduced. These instructions can only be glitched in the fetch phase.

Conditional branch instructions

Using the technique for evaluating near-jumps discussed in section 6.3.2 the conditional branch instruction `BREQ` can be evaluated.

The register pair R28:R29 was chosen to increment the preceding block, and the register pair R30:R31 was chosen to increment the following block. The timing of the glitches indicates that the instruction fetch phase has been glitched.

There are two main occurrences: instruction appears not to execute or jump to an undefined location. Instructions that appeared not to execute were encountered in the other tests, while the jumping to undefined locations is thought to be fetch corruption of the immediate operand. The corruption of the immediate operand is backed by the results of the multiply glitches.

When the test was set up so that branch should be taken it was possible to not execute the branch instruction and to take a jump to a different location. When branching forward, a branch backwards never occurred. When branching backwards, sometimes a forward branch took place. This indicates that glitching the sign register to a one state is extremely hard.

Skip instructions

Skip instructions are test instructions that conditionally execute the next instruction, skip instructions are specific to Atmel microcontrollers. Skip instructions can be combined with far jumps to create far branches. Instructions that follow the skip instructions can be one or two words long.

Two different tests were performed. First, the test was setup in such a way that the branch was not taken. Glitching the skip instruction reliably executes the next instruction, indicating that the condition was reversed or the skip was not executed.

The second test was set up so that the branch was not taken. Glitching the skip instruction does not appear to influence program execution.

A glitch of this instruction can be combined with a glitch of the next instruction. This will increase the chance that the glitch is successful.

7.2.3 Memory instructions

The XMEGA has a RISC architecture and requires data in its registers before performing calculations. It does not have special addressing modes which are present in other architectures. The load and store instructions are needed to perform those computations and are therefore important.

Load from data space

The `LDS` instruction retrieves data from SRAM, transfering an 8-bit value into a register. On the XMEGA, the instruction takes 3 clock-cycles. Memory glitches can affect instruction operation in a number of ways. The test presented in section 6.3.2 is conducted.

The XMEGA has IO memory and EEPROM mapped into the same address space as SRAM. This means reading from an arbitrary location could result in either SRAM or IO memory being read. There are a number of other possibilities. It is possible that no actual read is performed and an unrelated value is returned. Next, the correct value could end up in an incorrect register. In addition, memory from an incorrect location could end up in the correct register or memory from an incorrect location could end up in the incorrect register.

When grouped by time, two separate Groups emerge: one with a glitch in the opcode-fetch resulting in a correct result and unrelated register corruption, a second one with a glitch in the address fetch, resulting in an incorrect result in the correct register.

The first case shows register corruption while the correct instruction is executed, ruling out the possibility that the instruction was incorrectly fetched or decoded.

The second case returns an incorrect result, varying between returning a value from the prepended background and returning the value 0x00. The fact that no values were received from the background block following the target, indicates that bits in the address are being cleared.

The 0x00 value can be explained in two ways: address bits are cleared and the value is fetched from IO memory or the value on the bus is corrupted. Given the previous hypothesis that the value is incorrectly read from a lower address indicates that the value wawas fetched from IO or EEPROM memory.

A load from SRAM takes 1 cycle longer than a load from IO or EEPROM memory.

Note that the Atmel instruction set summary states that EEPROM has its own address space while the XMEGA documentation states that EEPROM, IO and SRAM are all in the same address space.

The EEPROM could additionally be programmed with a different value.

Store to data space

The STS instruction transfers a register value to memory. The `STS` instruction takes three instruction cycles and is two words long. The test presented in 6.3.2 was conducted.

All available memory is initialised to known values. A target value is written to the sentinel location in the middle of this block. If the location of the target does not coincide with the location of the sentinel, an address corruption is detected.

There are three groups of glitches spaced one clock cycle apart. The hypothesis being that they map onto the fetch of the opcode, the fetch of the address and the writing of the variable.

The first group of faults cannot locate the target value in memory. Additionally, a different value than the original is later retrieved from memory. The value retrieved from memory is either the original value of the variable, or it is set to the value of a different register. This result supports the hypothesis that the fault is introduced into the instruction fetch phase. This is supported by timing and that the end result can be adequately explained.

In addition, another possibility presents itself; the first instruction was decoded as a single word instruction and the address operand was decoded as an instruction. In this case, the address operand was 0xD427, which decodes to `RJMP`. An opcode mapping sorted by opcode can be found in section 12. The resulting jump would have caused reset like behavior. A possible extension to this test is to use a specially constructed address which will be decoded to an easily identifiable instruction. The `ADD` instruction would be a good candidate.

The second group yields a number of different results. The first result is that the target is found at a different location. In this case, the address at which it is found is always lower than the value at which it should have been written. The sentinel is still intact. Other results have an unchanged sentinel and the target is not found in memory. This indicates that the target was corrupted or that the address fetch was glitched in such a way as to point outside of SRAM. Given the timing of this group, an incorrect address fetch is the most likely.

The third and final group has correct results and no corruptions, except for the status register. It is interesting to note that the global interrupt flag appears to have been set in roughly half the cases. The setting of the status register cannot be explained by fetch corruption, because the glitched instruction has correctly computed the result. Fetch corruption is also ruled out by the timing information, as this phase corresponds to the SRAM access cycle.

In most cases, the glitched `STS` results can be explained by faults introduced into the instruction fetch phase. These faults have the tendency to cause bits to be changed from one to zero.

7.3 Conclusion

The normal operation and potential faults of the XMEGA are discussed in section 5.3.

The established XMEGA electrical glitch profile is excellent for building a fault model. The XMEGA can be glitched using a single glitch cycle. This makes it significantly easier to analyse the results, because the point at which a glitch is introduced can be accurately pinpointed.

The results presented in this chapter show that the glitches which appear on the XMEGA can be explained, and to a degree, predicted.

All instructions could be glitched to not execute correctly. This can be explained by a fault in the instruction fetch phase. This implies that the opcode of the instruction has been corrupted.

Most of the other results can also be explained by faults introduced during the instruction fetch phase. The `MUL` and `ADD` instructions can both be glitched to use different registers as their operands. The used registers are always lower than the original ones. If the resulting instructions are reconstructed and compared to the original instructions, it becomes apparent that bits have a tendency to transition from one to zero.

The results for the flow control instructions follow a similar pattern: the operands can be glitched. The resulting faults indicate that one bits have transitioned to zero.

The multi-cycle store and load instructions follow the same basic pattern. However, there are two instruction fetch cycles that can be glitched. The first instruction fetch phase can be glitched to affect the opcode and the second phase can be glitched to affect the address operand. As was the case with the flow control instructions a tendency for one bits to transition to zero was observed.

In addition, a smaller subset of the results exhibit register corruption, which cannot be explained by instruction fetch glitches. These results will be hard to explain using software based methods.

Chapter 8

Attack model

The previous chapter detailed the fault model for the XMEGA64A3. Those results are needed when creating an attack model. The goal of the attack model is to describe how an application running on an XMEGA64A3 can be subverted in the best possible manner. The chapter will start with a small discussion of parts of programs which are commonly attacked. Methods to attack these pieces of code will then be presented.

Glitching attacks can target a number of different code constructs: increasing loops to read memory, shortening of loops to corrupt variables, and manipulation of flow control instructions to skip checks. These are explained in more detail in [20].

Cryptographic procedures are also commonly attacked; if the data that the algorithm operates on can be corrupted, key retrieval can be attempted. The Bellcore attack [21] is a famous example.

The XMEGA64A3 has a support for AES and DES. Attacking these encryption implementations has previously been shown to be effective [9].

8.1 Using multiple glitches

Voltage glitching the XMEGA is not a guaranteed success. Therefore, an intelligent strategy is to string together a number of glitches which independently and together will result in the desired state being reached. While this does complicate the construction of a successful attack its success ratio should increase.

8.1.1 Skip instructions

The skip instructions evaluated in section 7.2.2 allow a single instruction to be conditionally executed. The code in Figure 8.1 depicts such a case. When the registers R30 and R31 have been initialised to different values the `EOR` command is executed. When the `CPSE` instruction is targeted, it glitches in two ways: the instruction is skipped or the condition is reversed. Skipping the instruction has absolutely no impact on the state of the processor. If the condition is reversed the `EOR` operation is not correctly executed. This is a cumbersome way of skipping the conditional instruction. However, it is also possible to additionally target the `EOR` instruction itself. Doing so will increase the chances of successfully reaching a target state.

8.2 Multiply by zero

The multiply instruction can be glitched as is described in section 7.2.1. When the instruction is glitched, there is a high chance at least one of the operands will be decoded to a different register. This will result in the variable being

```
cpse r31,r30
eor r30,r30
```

Figure 8.1: example of a skip instruction

multiplied by a different register.

After a reset, all registers are set to zero. If the application does not use all the registers, there will be a high probability that at least some of the registers will be set to zero. A multiplication with a zero will of course result in a zero. Not only are registers initialised to zero, but that value is also preserved across function calls. The avr-iar ABI [22] specifies that certain registers must be restored to their original value. Therefore, the chance that at least one register will be set to zero will be extremely high.

8.3 Taking uncontrolled jumps

The flow control instructions which were evaluated in section 7.2.2 can also be used to exert control over the target application.

Unexpected branches can be taken. This means that if a branch would not normally have been taken, it can be glitched to take that jump. While this is a good result, a more interesting attack is possible.

The operand of a branch or jump can also be glitched. This causes uncontrolled jumps to different locations to be attempted. If counter measures like those described in [23] are used this method could be more effective. In a scenario where two checks occur right after each other, one checking the inverse of the other, both the checks would have to be corrupted. A wild jump might be able to jump past both the checks.

8.4 Loading and storing incorrect values

Another possible attack uses the results presented in section 7.2.3. The load operation can be glitched to retrieve memory from an incorrect location.

There are a number of incorrect location ranges which are interesting from an attacker's point of view: uninitialized EEPROM, uninitialized IO space and uninitialized SRAM.

The areas are interesting because they default to known values. EEPROM defaults to 0xFF. The IO space and SRAM both default to 0x00. Particularly this last value is interesting, it can be used to perform the cryptographic attacks presented in [23].

It is also possible to glitch the storing of a computation. This can be used to ensure that the result of a computation does not update a variable.

Chapter 9

MSP430 results

Initially, the MSP430 was chosen as the target for this project. The created fault model would, given sufficient time, be used to attack the BSL. Even though it ultimately did not prove to be a good target for this research there were a number of results. These results do give some insight into the problems that can be encountered when trying to create a glitch profile. The normal operation of the MSP430 and its expected faults are discussed in section 5.1.

9.1 Add sled

The 'add sled' presented in section 6.3.1 was used to try and obtain a glitch. Initially, only resets and uncorrupted output were observed. In an attempt to induce a glitch, the parameters for the electrical glitch were widened. Of primary importance is that Vcc voltage was lowered. The faults in figure 9.1 were detected. While the corrupted outputs appear to be valid, an important point has to be made: the final byte 0xBB is corrupted. This constant value is sent after the critical section. A UART error is suspected to be caused by the lowered voltage.

The faults which are reported can thus be caused by either a glitch in the processor or the UART. The Vcc line was supplied a voltage of around 1.8 volts during these tests. A possible explanation is that the computer's serial interface was not correctly reacting to the sent signals, rather than that incorrect signals were being transmitted by the target. If these voltage ranges are to be investigated, a 1.8 volt serial interface would be required.

9.2 Flash write block

During testing, an interesting phenomenon was observed: programming the MSP430 requires a higher voltage than the processor needs to operate. This means that the voltage can be lowered to a level where the device cannot be programmed but will still operate. This phenomenon was not further researched. It is thought that this method can be helpful in glitching certain programs that update flash; any values that are written will no be correctly stored.

```
Correct output:
00 0A 0D FF 00 D7 00 D7 BB

Corrupted outputs:
00 0A 0D FF 00 97 00 97 7B
00 0A 0D FF 00 D7 00 97 3B
00 0A 0D FF 00 97 00 97 3B
00 0A 0D FF 00 97 00 B7 7B
```

Figure 9.1: MSP430 UART faults during an 'add sled' test

9.3 Voltage regulator

The MSP430x5xx family of devices have a regulator for core voltage. This voltage regulator has a high side voltage and a low side voltage. The high side voltage is fed from Vcc and the low side voltage is presented to the processor core of the MSP430 device. This can be measured using the Vcore line. Figure 9.2 shows the effects of a negative glitch on the Vcore line. The MSP430 reacts by increasing the voltage supplied to the core.

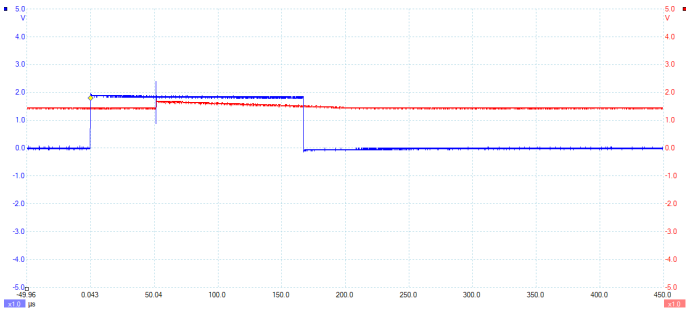

Figure 9.2: The red line represents the Vcore line and its reaction to a negative Vcc glitch

Chapter 10

Future research

As in any research, a number of avenues were left unexplored. This chapter details a number of such notable and interesting avenues.

10.1 MSP430 glitching

The research for this paper was first targeted at the MSP430 family of microcontrollers. Voltage glitching proved unsuccessful. It is believed that these microcontrollers can be glitched by other methods. For instance, reset-line glitching. For the MSP430F5xx family of microcontrollers, there is another possibility: glitching the vCore line [24]. The Vcore line allows the measurement of the internal core voltage. Ordinary voltage glitching will probably yield no results. However, it is thought that putting a load on this line might induce a fault in the IC.

The MSP430 truly is a low power microcontroller. A possible research avenue is to investigate the use of a low power serial interface. These interfaces can run at 1.8V and might correctly read the MSP430 UART at very low voltages.

Chapter 11

Conclusion

The goal of the research was to create a methodology that can be used to gain insight in microcontroller faults. Voltage glitching is a proven attack method and was chosen as the attack method for this research. The most common methods used to induce these faults target the power pin (Vcc), the reset pin or the clock pin. The pins can be attacked in a number of ways: short voltage dips, short voltage spikes and prolonged voltage dips. For this research, short voltage dips that are introduced in the Vcc line.

Next, the different ways in which induced faults or glitches affect operation of the microcontroller were enumerated. Prior to discussing these faults, the normal operation of a number of logical components of the MSP430 and the XMEGA were discussed, as were the ways in which these components can fail. The faults that presented themselves were categorised into memory and instruction faults. Instruction faults encompass the errors which may be introduced during the stages of instruction execution: fetch, decode, execute and store. Memory faults are divided into memory, memory address bus and memory data bus errors.

The possible effects were used as the basis for a number of tests that help determine the ways in which the processor is affected. Instrumentation that gives insight into the state of the microcontroller was developed, allowing different tests to be devised. However, by using only software methods it is not possible to conclusively attribute faults to any single component. Therefore, a domain model was presented consisting of memory, memory bus and instruction execution.

The first tests were concerned with acquiring a handle on the glitch. The timing and characteristics of the electrical glitch were improved. On the XMEGA, the glitch can fit inside a single clock cycle. After this, the timing relationship between glitch introduction and fault was investigated. Single instructions could now be targeted.

Further tests targeted single instructions and were divided into three groups: ALU instructions, flow control instructions and memory instructions. These tests give insight into the ways in which the processor was affected.

After performing these tests, the actual modified behaviour and its causes could be analysed. A pattern emerged in the origin of the faults. Most of the results fall into two groups: operand corruption and instruction not executed. Both results are seen in all three instruction groups. Timing data indicates that faults occur in the instruction fetch phase.

When a register operand for an instruction was corrupted, there was a tendency for the bits to transition from one to zero. This form of operand corruption was seen in all three instruction groups. For flow control instructions, the jump length was affected and for memory instructions the address could be glitched.

The other possibility was that the opcode of the instruction was corrupted. This means that a different instruction was executed. When the opcode was corrupted to a value that did not correspond to an instruction, it became an illegal or undefined instruction. Illegal instructions were treated as no-operation instructions on the XMEGA. The taking of unexpected branches was probably due to opcode corruption. Apart from instruction fetch faults, register corruption was also detected.

The resulting fault model of the XMEGA was used to describe an attack model. This attack model was presented to illustrate the main goal of fault models: exerting control on running code. To conclude, faults injected into the power line of a microcontroller can be considerable and, to a degree, predicted. The presented attack model is not exhaustive, but illustrates how disassembly can be analysed for glitching targets.

Bibliography

[1] E. Casey. *Handbook of digital forensics and investigation*. Academic Press, 2009.

[2] I. Breeuwsma et al. "Forensic imaging of embedded systems using JTAG (boundary-scan)". In: *Digital Investigation* 3.1 (2006), pp. 32–42.

[3] S. Skorobogatov. "Tamper resistance and physical attacks". In: *at Summer School on Cryptographic Hardware, Side-Channel and Fault Attacks* (2006).

[4] S.P. Skorobogatov. "Semi-invasive attacks-a new approach to hardware security analysis". In: *Technical report, University of Cambridge, Computer Laboratory* (2005).

[5] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*. Vol. 31. Springer-Verlag New York Inc, 2007.

[6] T. Goodspeed. "Practical Attacks against the MSP430 BSL". In: *Twenty-Fifth Chaos Communications Congress. Berlin, Germany*. 2008.

[7] T. Goodspeed and A. Francillon. "Half-blind attacks: mask ROM bootloaders are dangerous". In: *Proceedings of the 3rd USENIX conference on Offensive technologies*. USENIX Association. 2009, pp. 6–6.

[8] J. Balasch, B. Gierlichs, and I. Verbauwhede. "An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs". In: *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*. IEEE. 2011, pp. 105–114.

[9] I. Kizhvatov. "Side channel analysis of AVR XMEGA crypto engine". In: *Proceedings of the 4th Workshop on Embedded Systems Security*. ACM. 2009, p. 8.

[10] V. Pouget et al. "Tools and methodology development for pulsed laser fault injection in SRAM-based FPGAs". In: *8th Latin-American Test Workshop (LATW)*. Citeseer. 2007.

[11] Colby Hamilton. *Machine Casts Phantom Votes in the Bronx, Invalidating Real Ones*. URL: `http://www.wnyc.org/blogs/empire/2012/may/09/reports-find-machine-errors-led-uncounted-votes-2010/`.

[12] B. Dipert and L. Hebert. "Flash memory goes mainstream". In: *Spectrum, IEEE* 30.10 (1993), pp. 48–52.

[13] Riscure. *VcGlitcher*. URL: `http://www.riscure.com/benzine/documents/VCGlitcher.pdf`.

[14] Texas Instruments. *MSP430x5xx/MSP430x6xx Family User'sGuide*. URL: `http://www.ti.com/lit/ug/slau208j/slau208j.pdf`.

[15] Atmel. *XMEGA A3 Microcontroller*. URL: `http://www.atmel.com/Images/doc8068.pdf`.

[16] D.A. Patterson and C.H. Sequin. "RISC I: A reduced instruction set VLSI computer". In: *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press. 1981, pp. 443–457.

[17] A. One. "Smashing the stack for fun and profit". In: *Phrack magazine* 7.49 (1996), pp. 14–16.

[18] Atmel. *Instruction Set Manual*. URL: `http://www.atmel.com/Images/doc0856.pdf`.

[19] R.W. Hamming. "Error detecting and error correcting codes". In: *Bell System technical journal* 29.2 (1950), pp. 147–160.

[20] H. Bar-El et al. "The sorcerer's apprentice guide to fault attacks". In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382.

[21] A. Pellegrini, V. Bertacco, and T. Austin. "Fault-based attack of RSA authentication". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association. 2010, pp. 855–860.

[22] Atmel. *AVR034 : Mixing C and Assembly Code with IAR Embedded Workbench for AVR*. URL: http://www.atmel.com/Images/doc1234.pdf.

[23] F. Amiel, C. Clavier, and M. Tunstall. "Fault analysis of DPA-resistant algorithms". In: *Fault Diagnosis and Tolerance in Cryptography* (2006), pp. 223–236.

[24] Texas Instruments. *MSP430x5xxFamily*. URL: http://www.cizgi-tagem.org/resource/vfiles/tagem/dms_file/3457/slau208.pdf.

Chapter 12

Appendix

.

Opcode	instruction
0000 0000 0000 0000	NOP
0000 0001 dddd rrrr	MOVW
0000 0010 dddd rrrr	MULS
0000 0011 0ddd 0rrr	MULSU
0000 0011 0ddd 1rrr	FMUL
0000 0011 1ddd 0rrr	FMULS
0000 0011 1ddd 1rrr	FMULSU
0000 01rd dddd rrrr	CPC
0000 10rd dddd rrrr	SBC
0000 11dd dddd dddd	LSL
0000 11rd dddd rrrr	ADD
0001 00rd dddd rrrr	CPSE
0001 01rd dddd rrrr	CP
0001 10rd dddd rrrr	SUB
0001 11dd dddd dddd	ROL
0001 11rd dddd rrrr	ADC
0010 00dd dddd dddd	TST

Opcode	instruction
0010 00rd dddd rrrr	AND
0010 01dd dddd dddd	CLR
0010 01rd dddd rrrr	EOR
0010 10rd dddd rrrr	OR
0010 11rd dddd rrrr	MOV
0011 KKKK dddd KKKK	CPI
0100 KKKK dddd KKKK	SBCI
0101 KKKK dddd KKKK	SUBI
0110 KKKK dddd KKKK	ORI
0110 KKKK dddd KKKK	SBR
0111 KKKK dddd KKKK	ANDI
1000 000d dddd 0000	LD index Z
1000 000d dddd 1000	LD index Y
1000 001r rrrr 1000	STD
1001 000d dddd 0000 kkkk kkkk kkkk kkkk	LDS
1001 000d dddd 0001	LD index Z
1001 000d dddd 0010	LD index Z
1001 000d dddd 0100	LPM
1001 000d dddd 0101	LPM
1001 000d dddd 0110	ELPM
1001 000d dddd 0111	ELPM
1001 000d dddd 1001	LD index Y
1001 000d dddd 1010	LD index Y
1001 000d dddd 1100	LD
1001 000d dddd 1101	LD
1001 000d dddd 1110	LD
1001 000d dddd 1111	POP
1001 001d dddd 0000 kkkk kkkk kkkk kkkk	STS
1001 001d dddd 1111	PUSH
1001 001r rrrr 0100	XCH
1001 001r rrrr 0101	LAS
1001 001r rrrr 0110	LAC
1001 001r rrrr 0111	LAT
1001 001r rrrr 1001	STD
1001 001r rrrr 1010	STD
1001 001r rrrr 1100	ST
1001 001r rrrr 1101	ST
1001 001r rrrr 1110	ST
1001 0100 0000 1000	SEC
1001 0100 0000 1001	IJMP
1001 0100 0001 1000	SEZ
1001 0100 0001 1001	EIJMP
1001 0100 0010 1000	SEN
1001 0100 0011 1000	SEV
1001 0100 0100 1000	SES
1001 0100 0101 1000	SEH
1001 0100 0110 1000	SET
1001 0100 0111 1000	SEI
1001 0100 0sss 1000	BSET
1001 0100 1000 1000	CLC
1001 0100 1001 1000	CLZ
1001 0100 1010 1000	CLN
1001 0100 1011 1000	CLV
1001 0100 1100 1000	CLS
1001 0100 1101 1000	CLH
1001 0100 1110 1000	CLT
1001 0100 1111 1000	CLI
1001 0100 1sss 1000	BCLR
1001 0100 KKKK 1011	DES
1001 0101 0000 1000	RET

Opcode	instruction
1001 0101 0000 1001	ICALL
1001 0101 0001 1000	RETI
1001 0101 0001 1001	EICALL
1001 0101 1000 1000	SLEEP
1001 0101 1001 1000	BREAK
1001 0101 1010 1000	WDR
1001 0101 1100 1000	LPM
1001 0101 1101 1000	ELPM
1001 0101 1110 1000	SPM
1001 0101 1110 1000	SPM
1001 0101 1111 1000	SPM
1001 010d dddd 0000	COM
1001 010d dddd 0001	NEG
1001 010d dddd 0010	SWAP
1001 010d dddd 0011	INC
1001 010d dddd 0101	ASR
1001 010d dddd 0110	LSR
1001 010d dddd 0111	ROR
1001 010d dddd 1010	DEC
1001 010k kkkk 110k kkkk kkkk kkkk kkkk	JMP
1001 010k kkkk 111k kkkk kkkk kkkk kkkk	CALL
1001 0110 KKdd KKKK	ADIW
1001 0111 KKdd KKKK	SBIW
1001 1000 AAAA Abbb	CBI
1001 1001 AAAA Abbb	SBIC
1001 1010 AAAA Abbb	SBI
1001 1011 AAAA Abbb	SBIS
1001 11rd dddd rrrr	MUL
1010 0kkk dddd kkkk	LDS (16bit)
1010 1kkk dddd kkkk	STS
1011 0AAd dddd AAAA	IN
1011 1AAr rrrr AAAA	OUT
10q0 qq0d dddd 0qqq	LD index Z
10q0 qq0d dddd 1qqq	LD index Y
10q0 qq1r rrrr 1qqq	STD
1100 kkkk kkkk kkkk	RJMP
1101 kkkk kkkk kkkk	RCALL
1110 1111 dddd 1111	SER
1110 KKKK dddd KKKK	LDI
1111 00kk kkkk k000	BRCS
1111 00kk kkkk k000	BRLO
1111 00kk kkkk k001	BREQ
1111 00kk kkkk k010	BRMI
1111 00kk kkkk k011	BRVS
1111 00kk kkkk k100	BRLT
1111 00kk kkkk k101	BRHS
1111 00kk kkkk k110	BRTS
1111 00kk kkkk k111	BRIE
1111 00kk kkkk ksss	BRBS
1111 01kk kkkk k000	BRCC
1111 01kk kkkk k000	BRSH
1111 01kk kkkk k001	BRNE
1111 01kk kkkk k010	BRPL
1111 01kk kkkk k011	BRVC
1111 01kk kkkk k100	BRGE
1111 01kk kkkk k101	BRHC
1111 01kk kkkk k110	BRTC
1111 01kk kkkk k111	BRID
1111 01kk kkkk ksss	BRBC
1111 100d dddd 0bbb	BLD
1111 101d dddd 0bbb	BST
1111 110r rrrr 0bbb	SBRC
1111 111r rrrr 0bbb	SBRS

Figure 12.1: The opcode mapping sorted by opcode

```
Corrupted R30
SPH-2F SPL-D2 SR-00  R00-FF R01-00 R02-00 R03-00 R04-00 R05-00
R06-00 R07-00 R08-00 R09-00 R10-00 R11-00 R12-00 R13-00 R14-00
R15-00 R16-00 R17-20 R18-08 R19-00 R29-02 R21-00 R22-02 R23-5E
R24-60 R25-00 R26-07 R27-00 R28-04 R29-20 R30-F3 R31-00

Corrupted SREG R0 R30
SPH-2F SPL-D2 SR-03  R00-00 R01-00 R02-00 R03-00 R04-00 R05-00
R06-00 R07-00 R08-00 R09-00 R10-00 R11-00 R12-00 R13-00 R14-00
R15-00 R16-00 R17-20 R18-08 R19-00 R29-02 R21-00 R22-02 R23-5E
R24-60 R25-00 R26-07 R27-00 R28-04 R29-20 R30-F5 R31-00

Corrupted R0 R30
SPH-2F SPL-D2 SR-00  R00-80 R01-00 R02-00 R03-00 R04-00 R05-00
R06-00 R07-00 R08-00 R09-00 R10-00 R11-00 R12-00 R13-00 R14-00
R15-00 R16-00 R17-20 R18-08 R19-00 R29-02 R21-00 R22-02 R23-5E
R24-60 R25-00 R26-07 R27-00 R28-04 R29-20 R30-F5 R31-00
```

Figure 12.2: Multiple outputs of successfully glitched 'add sled' program

```
Correct output
SPH-2F SPL-B4 SR-14  R00-A0 R01-A1 R02-A2 R03-A3 R04-A4 R05-A5
R06-A6 R07-A7 R08-A8 R09-A9 R10-AA R11-AB R12-AC R13-AD R14-AE
R15-AF R16-50 R17-51 R18-52 R19-53 R29-54 R21-55 R22-56 R23-57
R24-58 R25-59 R26-5A R27-5B R28-5C R29-5D R30-FF R31-FF

Corrupted R30 - Glitch offset 0-22 ns end of glitch: 16-38 ns
SPH-2F SPL-B4 SR-14  R00-A0 R01-A1 R02-A2 R03-A3 R04-A4 R05-A5
R06-A6 R07-A7 R08-A8 R09-A9 R10-AA R11-AB R12-AC R13-AD R14-AE
R15-AF R16-50 R17-51 R18-52 R19-53 R29-54 R21-55 R22-56 R23-57
R24-58 R25-59 R26-5A R27-5B R28-5C R29-5D R30-FD R31-FF

Corrupted R30 - Glitch offset 432-520 ns end of glitch: 446-538 ns
SPH-2F SPL-B4 SR-14  R00-A0 R01-A1 R02-A2 R03-A3 R04-A4 R05-A5
R06-A6 R07-A7 R08-A8 R09-A9 R10-AA R11-AB R12-AC R13-AD R14-AE
R15-AF R16-50 R17-51 R18-52 R19-53 R29-54 R21-55 R22-56 R23-57
R24-58 R25-59 R26-5A R27-5B R28-5C R29-5D R30-FB R31-FF

Corrupted R30 - Glitch offset 924-986 ns end of glitch: 938-1006 ns
SPH-2F SPL-B4 SR-14  R00-A0 R01-A1 R02-A2 R03-A3 R04-A4 R05-A5
R06-A6 R07-A7 R08-A8 R09-A9 R10-AA R11-AB R12-AC R13-AD R14-AE
R15-AF R16-50 R17-51 R18-52 R19-53 R29-54 R21-55 R22-56 R23-57
R24-58 R25-59 R26-5A R27-5B R28-5C R29-5D R30-F7 R31-FF
```

Figure 12.3: Correct and corrupted 'or sled' program output

```
Correct output
Opcode: 1001 11 11 1111 1110  hamming distance: 0
SPH-2F SPL-B4 SR-01  R00-4B R01-EC R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB


Corrupted output: SR R0R1 = 0x3625 factors: 83 167 registers: r0,r16
Opcode: 1001 11 10 0000 0000  hamming distance: 5
SPH-2F SPL-B4 SR-00  R00-25 R01-36 R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB

Corrupted output: SR R0R1 = 0x3F47 factors: 97 167 registers: r2,r16
Opcode: 1001 11 10 0010 0000  hamming distance: 7
SPH-2F SPL-B4 SR-00  R00-47 R01-3F R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB

Corrupted output: SR R0R1 = 0x5953 instruction not correctly executed
Opcode: 0000 0000 0000 0000 (NOP)
SPH-2F SPL-B4 SR-00  R00-53 R01-59 R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB

Corrupted output: SR R0R1 = 0x4B8B factors: 83 233 registers: r0,r28
Opcode: 1001 11 01 1100 0000 hamming distance: 6
SPH-2F SPL-B4 SR-00  R00-8B R01-4B R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB

Corrupted output: SR R0R1 = 0x7c99 (factors: 167 191) registers: r16,r20
Opcode: 1001 11 11 0100 0000 hamming distance: 6
SPH-2F SPL-B4 SR-00  R00-99 R01-7C R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB
```

```
Corrupted output: SR R0R1 = 0x74c5 (factors: 167 179) registers: r16,r18
Opcode: 1001 11 11 0010 0000 hamming distance: 6
SPH-2F SPL-B4 SR-00  R00-C5 R01-74 R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB

Corrupted output: SR R0R1 = 0x6cf1 (factors: 167) registers: r16,r16
Opcode: 1001 11 11 0000 0000 hamming distance: 7
SPH-2F SPL-B4 SR-00  R00-F1 R01-6C R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB



Corrupted output: R0R1 = 0xc00b (factors: 211 233) registers: r24,r28
Opcode: 1001 11 11 1000 1100 hamming distance: 4
SPH-2F SPL-B4 SR-01  R00-0B R01-C0 R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB

Corrupted output: R0R1 = 0xC127 (factors: 197 251) registers: r22,r31
Opcode: 1001 11 11 1111 0110 hamming distance: 1
SPH-2F SPL-B4 SR-01  R00-27 R01-C1 R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB

Corrupted output: R0R1 = 0x9d37 (factors: 167 241) registers: r16,r30
Opcode: 1001 11 11 1110 0000 hamming distance: 4
SPH-2F SPL-B4 SR-01  R00-37 R01-9D R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB

Corrupted output: R0R1 = 0xdb59 (factors: 233 241) registers: r28,r30
Opcode: 1001 11 11 1100 1110 hamming distance: 2
SPH-2F SPL-B4 SR-01  R00-59 R01-DB R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB
```

Figure 12.4: Correct and corrupted outputs of glitched multiplications

```
Correct output:                 registers r16,r18
Opcode: 1001 11 11 0010 0000  hamming distance: 0
SPH-2F SPL-B4 SR-00  R00-C5 R01-74 R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB

Corrupted output: R0R1 = 0x3625 factors: 83 167 registers: r0,r16
Opcode: 1001 11 10 0000 0000  hamming distance: 2
SPH-2F SPL-B4 SR-00  R00-25 R01-36 R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB

Corrupted output: R0R1 = 0x6CF1 factors: 167  registers: r16,r16
Opcode: 1001 11 11 0000 0000  hamming distance: 2
SPH-2F SPL-B4 SR-00  R00-F1 R01-6C R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB

Corrupted output: R0R1 = 0x5953 instruction not correctly executed
Opcode: 0000 0000 0000 0000
SPH-2F SPL-B4 SR-02  R00-53 R01-59 R02-61 R03-65 R04-67 R05-6B
R06-6D R07-71 R08-7F R09-83 R10-89 R11-8B R12-95 R13-97 R14-9D
R15-A3 R16-A7 R17-AD R18-B3 R19-B5 R29-BF R21-C1 R22-C5 R23-C7
R24-D3 R25-DF R26-E3 R27-E5 R28-E9 R29-EF R30-F1 R31-FB
```

Figure 12.5: Results of a glitched `MUL` instruction, with operands containing 0

```
Correct output
SPH-2F SPL-B4 SR-14  R00-A0 R01-A1 R02-A2 R03-A3 R04-A4 R05-A5
R06-A6 R07-FA R08-A8 R09-A9 R10-AA R11-AB R12-AC R13-AD R14-AE
R15-AF R16-50 R17-51 R18-52 R19-53 R29-54 R21-55 R22-56 R23-57
R24-58 R25-59 R26-5A R27-5B R28-5C R29-5D R30-5E R31-5F

Corrupted output: SREG (carry set) R7 R0 (0x40) registers R0 R0
SPH-2F SPL-B4 SR-19  R00-40 R01-A1 R02-A2 R03-A3 R04-A4 R05-A5
R06-A6 R07-A7 R08-A8 R09-A9 R10-AA R11-AB R12-AC R13-AD R14-AE
R15-AF R16-50 R17-51 R18-52 R19-53 R29-54 R21-55 R22-56 R23-57
R24-58 R25-59 R26-5A R27-5B R28-5C R29-5D R30-5E R31-5F

Corrupted output: SREG (carry cleared) R7
instruction not correctly executed and status register set
SPH-2F SPL-B4 SR-18  R00-A0 R01-A1 R02-A2 R03-A3 R04-A4 R05-A5
R06-A6 R07-A7 R08-A8 R09-A9 R10-AA R11-AB R12-AC R13-AD R14-AE
R15-AF R16-50 R17-51 R18-52 R19-53 R29-54 R21-55 R22-56 R23-57
R24-58 R25-59 R26-5A R27-5B R28-5C R29-5D R30-5E R31-5F

Corrupted output: R7 R0 (0xF2) registers R0 R18
SPH-2F SPL-B4 SR-14  R00-F2 R01-A1 R02-A2 R03-A3 R04-A4 R05-A5
R06-A6 R07-A7 R08-A8 R09-A9 R10-AA R11-AB R12-AC R13-AD R14-AE
R15-AF R16-50 R17-51 R18-52 R19-53 R29-54 R21-55 R22-56 R23-57
R24-58 R25-59 R26-5A R27-5B R28-5C R29-5D R30-5E R31-5F

Corrupted output: R7 R4 (0xF6) registers R4 r18
SPH-2F SPL-B4 SR-14  R00-A0 R01-A1 R02-A2 R03-A3 R04-F6 R05-A5
R06-A6 R07-A7 R08-A8 R09-A9 R10-AA R11-AB R12-AC R13-AD R14-AE
R15-AF R16-50 R17-51 R18-52 R19-53 R29-54 R21-55 R22-56 R23-57
R24-58 R25-59 R26-5A R27-5B R28-5C R29-5D R30-5E R31-5F
```

Figure 12.6: Correct and corrupted output for the add test