On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms

Thomas Korak and Michael Hoefler
Institute for Applied Information Processing and Communications
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria.
Email: thomas.korak@iaik.tugraz.at, mi.hoefler@gmail.com

Abstract—**Forcing faulty outputs of devices implementing cryptographic primitives in order to reveal some secret information is a well-known attack strategy. Tampering with the clock signal or the supply voltage are two popular techniques for generating erroneous behaviour. In this work we perform an in-depth analysis of the vulnerability of two different microcontroller platforms on clock and supply voltage tampering. The influence on three different groups of instructions is discussed in detail: *arithmetical/logical instructions, branch instructions* and *memory instructions*. A novel approach, combining short-time underpowering with clock glitches, is applied in order to increase the reproducibility of the fault and the probability for the fault to occur. Results show that the *fetch stage* and the *execution stage* of the instruction pipeline are mainly affected by clock glitches, leading to skipped or duplicated execution or faulty calculation results. One sample per platform is used for the experiments, but for each fault type an interval for the parameters leading to the fault with a probability of 100 % is given. The values in the middle of this interval are less sensitive to sample distribution as well as environmental influences. This knowledge allows to efficiently attack software implementations of cryptographic primitives implemented on one of the evaluated platforms.**
Keywords: microcontroller, clock glitch, power glitch.

I. Introduction

Physical attacks are a popular choice for breaking cryptographic implementations. This type of attacks take advantage of small leakages in e.g., timing [17], power consumption [18], or electromagnetic emanation [13] in order to reveal a secret key. Fault attacks can also be classified as physical attacks. In the case of fault attacks, the attacker tries to extract some information about the secret value by enforcing some faulty behaviour in the attacked device. The faulty behaviour is generated by influencing environmental parameters. Tampering with the supply voltage [29], the clock signal [3], the temperature [15], or injecting EM pulses [20] can be categorized as non-invasive fault attacks. They typically do not require a modification of the device. Inducing faults due to laser light [27] are semi-invasive fault attacks as a depackaging of the chip is required. A comprehensive overview of fault-injection methods can be found in the work of Bar et al. [4]. In this work we focus on non-invasive fault attacks tampering with the clock signal and the supply voltage targeting two different kinds of microcontroller units (MCUs).

In order to conduct the aforementioned attacks, physical access to the attacked device is required. For non-invasive attacks, the device can be put in place after the attack, so the probability, that the attack remains undetected is high. Sensor nodes are potential targets for physical attacks. They often operate in hostile environments and are only sporadically observed by human beings. This fact makes it easy for an attacker to remove the node for the duration of the attack and put it in place afterwards. For the analyses in this work we have chosen two MCUs frequently used as central processing unit in sensor nodes, one Atmel ATxmega 256 and one ARM Cortex-M0.

A. Related Work

A huge number of non-invasive fault attacks on cryptographic primitives have been reported in literature during the last years. Popular attack targets are hardware as well as software implementations of block ciphers and software implementations of the RSA algorithm. Next to the attacked algorithm, one can differ whether the datapath (e.g. change intermediate values in the state of a block cipher) or the control logic (e.g. influence the round counter) is affected by the fault.

Selmane et al. [24] have shown, how setup-time violations due to underpowering can be used to attack the Advanced Encryption Standard (AES) block cipher. In [5], the authors present successful fault attacks targeting an AES and a RSA software implementation. The implementations run on an ARM 9 general purpose CPU and underpowering is used to inject the faults. The authors of [1] apply an FPGA implementation of AES to perform fault attacks using clock glitches. Practical results have been used in order to verify the theoretical analyses. Practical fault attacks targeting six different block ciphers implemented on an LSI have been reported by Fukunaga, Toshinori and Takahashi in [12]. The authors have used clock glitches in order to inject faults.

Choukri and Tunstall take advantage of power supply glitches in order to modify the round counter of a secret-key algorithm in [8]. A similar approach to modify the AES round counter is presented in [10]. Here, the authors use electromagnetic pulses in order to enforce the faulty behaviour.

Koemmerling and Kuhn [19] present ways to extract secret data from smart cards. They cover a wide range of physical attacks and also propose some countermeasures.

Schmidt et al. [23] use spikes in the power supply to attack an RSA implementation. Kim and Quisquater [16] show that with two precisely timed power glitches the countermeasures of a secured RSA implementation can be circumvented.

All the above mentioned works either attack a hardware implementation or a specific software implementation. Balasch et al. [3], in contrast, observe the influence of clock glitches on the instruction set of a microcontroller in a black-box scenario. They show, that with different shapes of clock glitches, wrong instructions can be executed.

B. Contribution

In this work we follow a similar approach like the authors in [3]. The main difference is, that we compare the influence of clock glitches targeting selected instructions for two different MCU types, one ATxmega 256 and one ARM Cortex-M0. In [3], an ATmega 162 MCU serves as device under test. Using two different MCU platforms allows us to draw comparisons about the influence of similar fault injections (one sample per MCU platform has been used for the conducted experiments). Besides the usage of two different MCU platforms, we provide the following, novel contributions:

- We are the first to discuss the effects of similar faults on two different pipeline architectures, the three-stage pipeline of the ARM Cortex-M0 and the two-stage pipeline of the ATxmega 256. Results show that on both platforms the *fetch stage* and the *execute stage* are affected by clock glitches.
- To the authors knowledge, this is the first work where short-time underpowering of the attacked device is applied in addition to clock glitches in practical fault attacks. This approach has figured out to significantly increase the efficiency as well as the reproducibility of fault attacks.
- Due to the fact that underpowering can increase the efficiency of clock glitches, the brown-out detection might be used as a simple countermeasure. Therefore, the effect of short-time underpowering on the brown-out detection has been evaluated. Results show that for well-chosen values for supply voltage reduction and duration, underpowering is not detected by the brown-out detection mechanism on the ARM Cortex-M0 with a given probability.
- The effects of fault injections on instructions from three different groups are discussed and compared in detail for both evaluated MCU platforms: *arithmetical/logical instructions*, *branch instruction*, and *memory instructions*.
- For each investigated instruction, an interval for the parameters leading to a specific erroneous behaviour with 100 % probability, is given. Giving such an interval allows to compensate for sample distribution as well as environmental influences.

C. Outlook

The rest of the work is structured as follows. In Section II we give an overview of fault injection methodologies with the focus on clock glitches and underpowering. Section III introduces the setup for the fault attacks, the two microcontrollers as well as the test scenarios. In Section IV the results are discussed. A comparison of our results with previous related work is done in Section V and Section VI concludes the paper.

II. Fault Injection Methodology

In the following section we are going to shortly discuss the different types of fault injection mechanisms. After the general discussion we put the focus on clock glitches and underpowering, because these fault injection approaches have been used for the attacks presented in this work.

When talking about fault attacks, one first has to differ between non-invasive, semi-invasive and invasive fault attacks:

Non-invasive fault attacks. This type does not require a modification of the attacked device in order to induce the fault. Therefore, the probability that the attack remains unrecognised by the victim, is high. Furthermore, no sophisticated equipment is required for this kind of attacks. The use of electromagnetic pulses (e.g., [9]), positive or negative spikes in the power line (e.g., [29]) or adding additional clock edges (e.g., [3]) are typical methods. Here, the attacker tries to influence the timing behaviour of the attacked device. If the critical path of the combinational logic of the device is undercut, the probability for inducing faulty behaviour is high. EM pulses or operating the device outside the allowed temperature range (e.g., [22], [25]) might further lead to faults in memory.

Semi-invasive fault attacks. In order to perform a semi-invasive fault injection, a penetration of the attacked device is required. In the case of optical fault attacks, this modification equals a de-packaging of the chip to get access to the chip die, but the passivation layer stays intact. Laser light (e.g., [28]) as well as flash light (e.g., [27]) can be used to induce a fault. A laser allows to target specific bits in memory or registers while flash lights only allow global faults across a larger region of the chip.

Invasive fault attacks. A penetration of the attacked device is also required for this kind of attack. After the de-packaging step also the passivation layer is removed in order to access the metal layer with microprobes (e.g., [26]). This approach allows targeting single bus lines on the chip and modifying their value (e.g., *stuck-at-zero, stuck-at-one*). Invasive fault attacks require the most sophisticated equipment with the advantage, that nearly no limitations for fault injections exist.

In order to take advantage of an erroneous output of a cryptographic operation, several analysis techniques have been presented in the last years: Differential Fault Analysis (DFA) [6], Collision Fault Analysis (CFA) [14] and Ineffective Fault Analysis (IFA) [7]. These fault analysis methods are out of the scope of this work and will therefore not be discussed in detail.

A. Clock Glitches

As already discussed, clock glitches can be categorized as non-invasive attacks. They can be applied to devices, which are supplied with an external clock signal. The goal is to violate the timing constraints of the device by adding additional clock edges. Figure 1 shows one clock signal, where an additional clock edge is inserted. T equals the clock period during normal operation, while T_{Glitch} equals the clock period in the case of a fault injection. Figure 2 depicts the clock signal generated with our fault board, measured with an oscilloscope. In fact, clock signals with T_{Glitch} between 8 ns and 22 ns are plotted

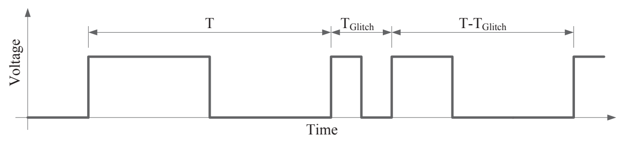

Fig. 1. Violating the critical path delay by inserting an additional positive clock edge.

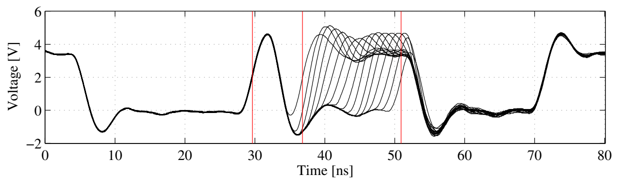

Fig. 2. Clock signal with different settings for T_{Glitch} between 8 ns and 22 ns.

overlaid. Further information about the generation of the clock signal and the fault attack board can be found in Section III.

For devices, which are supplied with an external clock, a maximum clock frequency f_{max} is defined typically. The minimum period T_{min} equals the reciprocal of f_{max}. As long as $T_{Glitch} \geq T_{min}$, no erroneous behaviour can be observed, but if $T_{Glitch} < T_{min}$, the probability for observing an erroneous behaviour increases. The reason for the erroneous behaviour is a timing violation. Each combinational path in a circuit has a specific propagation delay (T_P), which describes the time interval between changing the input and providing a valid output value. Registers at the output of the combinational logic block sample the value, typically at the positive clock edge. If the positive clock edge arrives before the output of the combinational logic has settled to a stable value due to a timing violation enforced by a clock glitch ($T_{Glitch} < T_P$), the registers store erroneous values leading to faulty computations as a consequence.

B. *Underpowering*

Tampering with the supply voltage can be categorized as non-invasive fault attack. Underpowering means to reduce the value of the supply voltage of the attacked device below the minimum value, the device is specified for. Depending on the duration of the reduction of the supply voltage, one can distinguish between permanent underpowering (e.g., for the whole duration of an algorithm execution) and transient underpowering (e.g., only for some clock cycles). The latter approach might remain undetected by some detection mechanisms, as also shown in our results. Underpowering increases the propagation delay T_P of combinational logic (for a detailed explanation of the effect of the supply voltage on the propagation delay, the authors refer to [29]). Due to this fact, similar effects can be achieved when tampering with the clock signal and the supply voltage. That means, when reducing the supply voltage, timing violations can be enforced. In the result section, a combination of clock glitches and underpowering is presented, which leads to an improved fault injection. The supply voltage is only reduced during the clock glitch for that purpose.

III. Fault Injection Setup and Evaluation Scenario

In the following section the setup for the fault injection experiments is introduced. An FPGA-based fault board was used for tampering with the clock signal as well as the power supply of the microcontroller. Next, the two investigated microcontrollers, one ARM Cortex-M0 and one ATxmega 256 are introduced. At the end of this section, an explanation of the conducted test scenarios is given.

A. *Fault Board and Extension Boards*

The device we have used in order to tamper with the supply voltage and the clock signal, is a custom-made fault board. The fault board is depicted in Figure 3. The main part of this fault board is a *XILINX Spartan-6 XC6SLX45* FPGA. For communication with the control computer, the fault board is equipped with a USB port and the configuration is done via MATLAB scripts on the control computer. A huge number of IO pins for connecting a wide range of devices are available. For the experiments performed in this work, three pins are essential: the *clock signal*, the *supply voltage*, and the *trigger signal*. A block diagram showing the setup can be found in Figure 4. For visualizing the clock signal and the supply voltage, a *PicoScope 5203* from *Pico Technology*[1] has been used.

Clock glitch generation. On the fault board, a similar approach like presented in [1], [11] is applied to generate the clock signal and insert glitches into it. For all the experiments we have used a nominal clock frequency of 24 MHz ($T \approx 41.7$ ns) in order to clock the MCUs. Values for T_{Glitch} between 5.9 ns and 22.0 ns can be achieved with this setup, which equals a frequency range between 45 MHz and 170 MHz. Figure 2 depicts clock signals generated with the fault board. T_{Glitch} values in the range of 8.0 ns up to 22.0 ns, with a step size of 1.0 ns, were measured for creating this plot. The glitch is inserted after a trigger event on a predefined pin of the FPGA. Defining the number of clock cycles between the trigger event and the glitch insertion allows to precisely select the point in time for the glitch to occur.

Supply voltage manipulation. A multiplexer with different, configurable voltage values at its inputs allows to tamper with the supply voltage of the attacked device. Voltage values in the range between 0 V and 5 V are supported. For underpowering, we have set $U_1 = 3.3\,V$ and $U_{Glitch} = (3.3 - U_{diff})\,V$. U_1 equals the voltage at multiplexer input 1 and U_{Glitch} equals the voltage at multiplexer input 2, respectively. U_{diff} defines the reduction of the supply voltage during underpowering. The supply voltage reduction takes place a defined number of clock cycles after a trigger event and also the duration of the underpowering can be defined precisely.

Extension boards. Figure 3 also depicts the extension boards for the investigated microcontrollers discussed in the following. The extension boards allow to attack a wide range of devices using the same fault attack board.

[1]http://www.picotech.com/

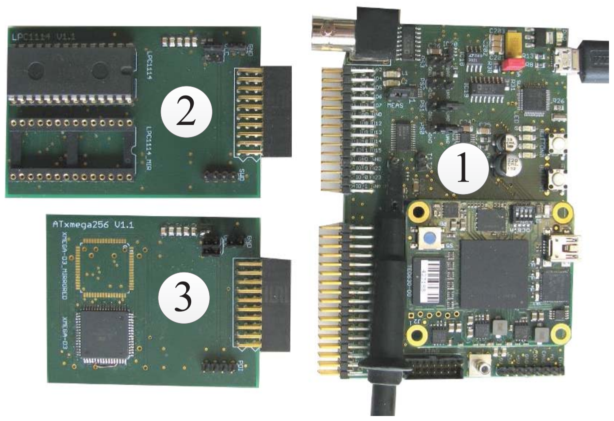

Fig. 3. (1): Fault board; (2): ARM Cortex-M0 extension board; (3): ATxmega 256 extension board.

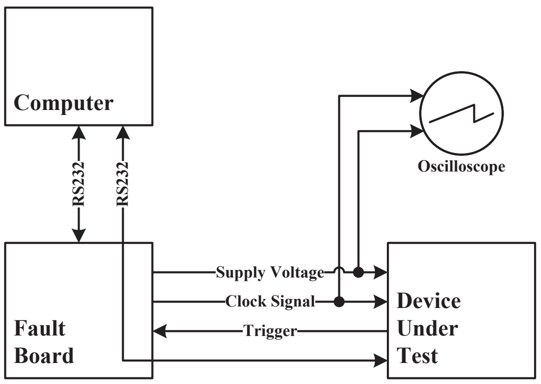

Fig. 4. Block diagram of the fault injection environment.

B. Investigated Microcontrollers

1) ARM Cortex-M0: The ARM Cortex-M0 represents the lower end of the ARM Cortex-M family. It is a 32 bit RISC processor with Von-Neumann architecture. It supports 56 instructions, most of them belonging to the ARM *Thumb* instruction set. For instruction execution the ARM Cortex-M0 applies a three-stage pipeline as depicted in Figure 5 with the three stages: *Fetch stage*, *decode stage*, and *execute stage*. Most operations of the ARM Cortex-M0 are performed with 16-bit instructions. As the CPU provides a 32-bit bus system, in every second cycle two 16-bit instructions are fetched in parallel.

For the experiments performed in this work, NXP's LPC 1114 implementation of the ARM Cortex-M0 has been used [21]. This implementation supports a maximal clock frequency f_{max} of 50 MHz and a supply voltage range between 1.8 V and 3.6 V. For the performed attacks on the ARM Cortex-M0, a nominal clock frequency of 24 MHz is used, which equals a factor of 0.48 compared to f_{max}. When using clock glitches with the maximal clock glitch frequency of 170 MHz the maximal operating frequency f_{max} is exceeded by factor of 3.40. A nominal supply voltage of 3.3 V is used during the practical experiments on the ARM Cortex-M0.

2) Atmel ATxmega 256: The ATxmega 256 is a 8/16 bit microcontroller for low-power applications with a RISC architecture. The separated memories for data and program code make the ATxmega 256 a CPU with Harvard architecture. It has a two-stage instruction pipeline with one *fetch stage* and one *execute stage* as shown in Figure 6. During the execution of an instruction, the next instruction is simultaneously loaded from the program memory. Using a 16-bit program memory bus allows to load a 16-bit instructions in a single clock cycle. The ATxmega 256 supports 142 Atmel AVR instructions, most of them executing in a single clock cycle. For further information the authors refer to the datasheet of the microcontroller [2].

For our practical experiments we use the ATxmega 256A3 which supports a maximal operating frequency f_{max} of 32 MHz. The supply voltage range is specified between 1.6 V and 3.6 V. When using a clock frequency above 12 MHz at least 2.7 V are recommended. For the attack experiments performed in this work, the ATxmega 256 operates on a nominal clock

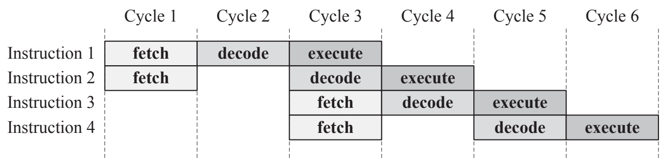

Fig. 5. Pipeline diagram showing the instruction execution procedure of the ARM Cortex-M0.

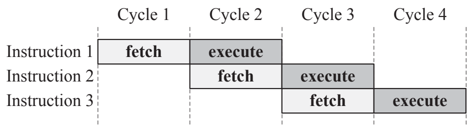

Fig. 6. Pipeline diagram showing the instruction execution procedure of the ATxmega 256.

frequency of 24 MHz, which equals a factor of 0.75 compared to f_{max}. The maximal clock glitch frequency of 170 MHz exceeds the maximal operating frequency f_{max} by factor of 5.31. For all attacks a nominal supply voltage of 3.3 V is used.

C. Test Scenarios

There are two important characteristics which have to be considered when analysing the possible influences and the erroneous behaviour of microcontrollers to an injected fault. First, the execution process of an instruction is separated in several CPU stages (instruction pipeline) and second, results may vary depending on the T_{Glitch} value used for the attack. Consequently, a fault in the instruction fetch stage could possibly result in different or wrong instructions either by reading from a wrong memory address or by reading faulty data. Similar effects are possible if the instruction is fetched correctly but misinterpreted in the decode stage. In the execution stage a fault could result in wrong calculations and wrongly or non-updated registers. To be able to draw meaningful conclusions from the resulting behaviour after an attack the clock period is only tampered within one pipeline stage. Further, we focus on three classes of instructions: *arithmetical/logical instructions*, *branch instructions* and *memory instructions*. According to these categories we have chosen a set of instructions for both microcontrollers, the ARM Cortex-M0 and the ATxmega 256, summarized in Table I.

TABLE I. OBSERVED INSTRUCTIONS.

Instruction Class	ATxmega 256	ARM Cortex-M0

Arithmetical/Logical	`add Rd,Rn`
	`mul Rd,Rn`
		`adds Rd,Rn`
		`muls Rd,Rn`
		`lsls Rd,#imm`

Branch	`breq label`	`beq label`

Memory	`ld Rd,X`
	`st X,Rn`
		`ldr Rd,[Rn]`
		`str Rd,[Rn]`

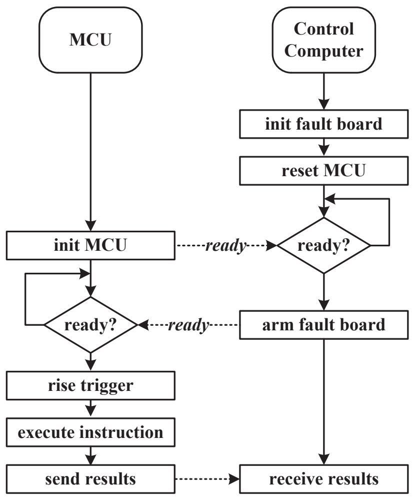

Fig. 7. Sequence of operations performed during an attack.

In detail the chosen instructions are arithmetical addition, multiplication and the logical shift operations for the group of *arithmetical/logical instructions*. To illustrate the effects on *branch instructions* we use a conditional branch instruction and as examples for *memory instructions*, load and store operations are observed.

For each instruction a test application is written in mixed C and assembly language where inline assembly is used to initialize the general purpose registers and to define the attacked instruction. To ensure proper execution of all other code parts and to avoid any unpredictable side effects, the attacked instruction is surrounded by `nop` instructions. Figure 7 illustrates the sequence of operations performed during an attack by the MCU and the control computer. In detail, each test application complies with the following attack procedure.

(1) Initialization of the microcontroller. In the initialization routine of the application, general microcontroller configurations are performed. This comprises the system clock setup for using the clock signal provided by the fault board, the I/O port configuration for the trigger, and the UART interface initialization.

(2) Synchronization with the control computer. The synchronization during an attack between the fault board and the test application is done with UART commands, exchanged between the microcontroller and the control computer. After the initialization is finished, the microcontroller signals this state, which in turn allows the computer to prepare the fault board for an attack. Once the fault board is ready and sensitive to the trigger input, the execution on the microcontroller is resumed.

(3) Rise trigger pin. To ensure a cycle accurate synchronization between the fault board injecting the glitch and the microcontroller performing the instruction which should be hit by the attack, the microcontroller rises its trigger output signal. A predefined number of clock cycles after the trigger event, the glitch is injected.

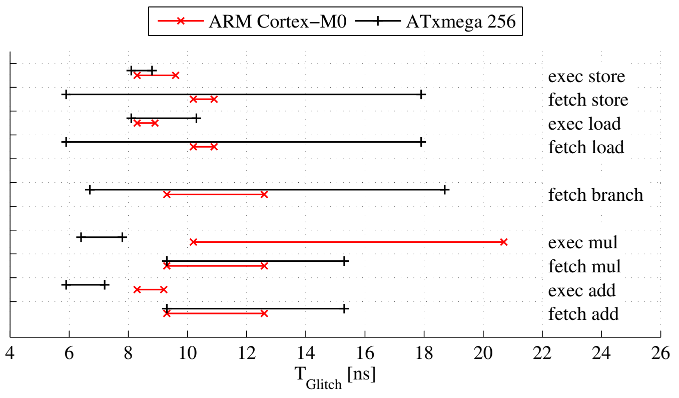

Fig. 8. Summary of the T_{Glitch} values leading to successful attacks for both investigated microcontrollers and all the observed instructions.

(4) Execution of the attacked instruction. The affected instruction is then executed in compliance with the time interval between the trigger event and the glitch injection. Additionally, `nop` instructions are used to keep this time interval constant and independent from other necessary assembly instructions executed between the trigger event and the attacked instruction.

(5) Result communication. In the final step, all accessible CPU and general purpose register values are transferred to the control computer and finally compared to the result of a reference execution without fault injection. For each instruction, the impact of the fault at the different execution stages (fetch, decode, execute) and for varying T_{Glitch} values is evaluated. The captured data sets of erroneous results allow to make assumptions about the internal CPU instruction execution process.

IV. RESULTS

This section describes in detail the obtained erroneous behaviour of the ARM Cortex-M0 and ATxmega 256 in consequence of injected faults. For both microcontrollers we present common and individual results for all attacked instructions listed in Table I. We further evaluate the impact of the attacked execution stage as well as the value of the glitch period T_{Glitch} on the instructions. All the achieved results presented in this section are visualized and summarized in Figure 8. This figure depicts the intervals for T_{Glitch} for each instruction and the two affected pipeline stages for the individual microcontrollers. It is notable that for all the values inside the intervals the probability for the fault occurrence was 100 %. For values close to the interval borders, but outside the interval, faults could also be observed, but with a smaller probability of occurrence.

A. ARM Cortex-M0 Specific Results

First attacks targeting the ARM Cortex-M0 figured out that this MCU type is rather insensitive to clock glitch attacks. Regardless of the used system clock frequency and the glitch period T_{Glitch}, only a negligible number of attacks led to mainly non-reproducible results. In order to increase the impact of forced timing violations caused by injected clock glitches we used an approach of reducing the supply voltage level for a short period during the clock glitch attack. As illustrated in Figure 9, the supply voltage of the ARM Cortex-M0 is reduced to 1.2 V, this value is beyond the minimum operating voltage of 1.8 V given in the datasheet [21]. The short-time underpowering in combination with the clock glitch enables successful fault injection with reproducible results. Applying underpowering without a clock glitch did not lead to a faulty behaviour of the device. Injecting faults only by underpowering was not the aim of this work, so we did not further investigate this.

In contrast to only inserting clock glitches, additional underpowering might be detected by an optional applicable brown-out detection provided by the ARM Cortex-M0. Therefore, we analysed the response characteristic of the brown-out detection for its most sensitive configuration, detecting supply voltage drops below a voltage level of $U_{Lvl} = 2.8$ V. The ARM Cortex-M0 provides two strategies for handling detected voltage drops. First, the brown-out interrupt and second, the brown-out reset. In our tests, the brown-out interrupt service routine was correctly executed for the interval $1.0\,V \leq U_{Glitch} < U_{Lvl}$ using an underpowering duration as shown in Figure 9. For $U_{Glitch} < 1.0$ V instead of executing the interrupt service routine, the actual program flow continued with a probability of about 10%. In about 90% of the test cases, a hard fault exception[2] occurred. In contrast to the brown-out interrupt, the brown-out reset was correctly performed in all cases where $U_{Glitch} < U_{Lvl}$. However, in both cases the brown-out detection responds to voltage drops only if the duration includes at least two rising clock edges.

In a further experiment, we have analysed the relationship between U_{Glitch} and T_{Glitch} when attacking the fetch stage and the execution stage of the pipeline, respectively. Figure 10 depicts the result of this experiment for the `adds Rd,Rn` instruction and Table II summarizes the corresponding values.

TABLE II. SUMMARY OF THE RELATIONSHIP BETWEEN U_{Glitch} AND THE INTERVAL FOR $T_{Glitch} = [t_{start}, t_{end}]$.

	Execute			Fetch		
U_{Glitch}	t_{start}	t_{end}	Δ	t_{start}	t_{end}	Δ
V	ns	ns	ns	ns	ns	ns
1.50	-	-	-	7.20	7.35	0.15
1.40	7.55	7.75	0.20	7.55	8.25	0.70
1.30	8.10	8.55	0.45	7.90	9.50	1.60
1.20	8.10	8.85	0.75	9.00	12.55	3.55
1.10	8.75	10.70	1.95	9.30	12.55	3.25
1.00	9.95	17.68	7.73	9.80	12.85	3.05
0.90	12.35	14.40	2.05	-	-	-

Attacks targeting the execution stage succeed in the range of $0.9\,\text{V} \leq U_{Glitch} \leq 1.4\,\text{V}$. The increasing interval size for the T_{Glitch} values with decreasing U_{Glitch} values is clearly visible. For $U_{Glitch} = 1V$, T_{Glitch} values in the interval $[9.95, 17.68]$ ns produce faults, for $U_{Glitch} = 1.4V$, T_{Glitch} values in the interval $[7.55, 7.75]$ ns produce faults. The case for $U_{Glitch} = 0.9V$ has to be discussed separately, here the interval is getting smaller again, non-compliant with the observed trend. With this setting, several hard-faults happened due to the low voltage. Therefore a 100 % probability for the fault injection could only be reached in a small interval.

Attacks targeting the fetch stage succeed in the range of $1.0\,\text{V} \leq U_{Glitch} \leq 1.5\,\text{V}$. The increasing interval size for the T_{Glitch} values with decreasing U_{Glitch} values is clearly visible. For $U_{Glitch} = 1V$, T_{Glitch} values in the interval $[9.80, 12, 85]$ ns produce faults, for $U_{Glitch} = 1.5V$, T_{Glitch} values in the interval $[7.20, 7.35]$ ns produce faults. One interesting observation is, that for $U_{Glitch} = 1.2V$, the T_{Glitch} intervals which affect the fetch and the execution stage do not overlap. This fact gives an attacker the opportunity to attack either the fetch stage (e.g. with $T_{Glitch} = 11ns$) or the execution stage (e.g. with $T_{Glitch} = 8.5ns$) in the same clock cycle.

For the rest of this work, when clock glitch attacks on the ARM Cortex-M0 are discussed, underpowering is also always applied with deactivated brown-out detection and $U_{Glitch} = 1.2V$.

B. *Atmel ATxmega 256 Specific Results*

Compared to the ARM Cortex-M0, the ATxmega 256 is more sensitive to clock glitches. Underpowering was not required in order to successfully inject faults. Due to this fact we did not further investigate the brown-out detection on this MCU. According to its datasheet, the typical brown-out detection time[3] t_{BOD} is 400 ns in continuous mode and 1 ms in sampled mode, respectively. This is far above the time we require for a short-time underpowering meaning that it would not be detected by the brown-out detection.

C. *Arithmetical/Logical Instructions*

During the attack experiments we observed a relation between the erroneous results and the attacked execution stage of the pipeline. This applies for both microcontrollers, only depending on their specific pipeline structure shown in Figure 5 for the ARM Cortex-M0 and in Figure 6 for the ATxmega 256, respectively.

1) ARM Cortex-M0: To illustrate the effects of injected faults concerning the group of *arithmetical/logical instructions* on the ARM Cortex-M0, we first attacked a single `adds Rd,Rn` instruction conforming the attack scenario described in Section III-C. The sequence of executed instructions and the corresponding position within the CPU pipeline is depicted in Table III.

TABLE III. ATTACK ON A SINGLE `ADDS RD,RN` INSTRUCTIONS ON THE ARM CORTEX-M0 MICRCONTROLLER.

Instruction	**Cycle**							
	i	***i+1***	***i+2***	***i+3***	***i+4***	***i+5***	***i+6***	***i+7***
`nop`	F	D	E					
`nop`	F		D	E				
`nop`			F	D	E			
`adds Rd,Rn`			F		D	E		
`nop`					F	D	E	
`nop`					F		D	E

Fetch stage. When inserting a glitch with a glitch period T_{Glitch} between 9.3 ns and 12.6 ns in clock cycle *i+2* the

[2]A hard fault is an exception with the highest priority. If a hard fault happens, the fault handler is executed.

[3]t_{BOD} equals the duration, the supply voltage has to be below the predefined voltage level, before a brown-out interrupt is executed.

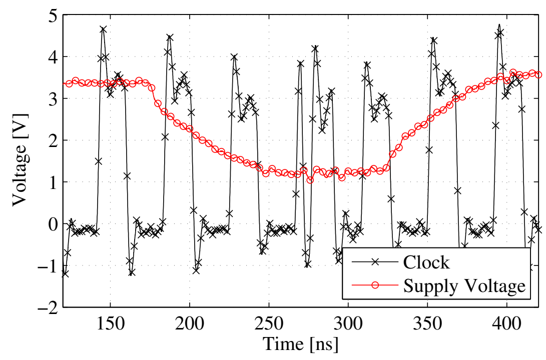

Fig. 9. Clock signal and supply voltage during an attack ($T_{Glitch} = 10.0$ ns, $U_{Glitch} = 1.2$ V).

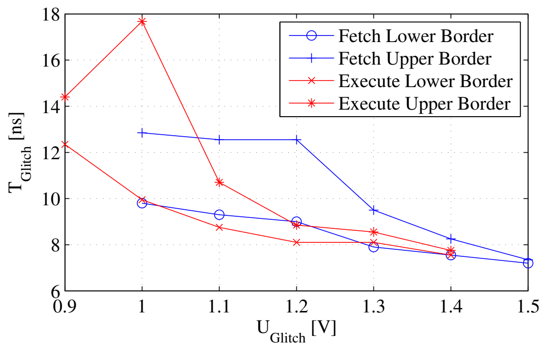

Fig. 10. Relationship between T_{Glitch} and U_{Glitch} for attacks to succeed.

instruction `adds Rd,Rn` is attacked during the fetch process and subsequently not executed. Considering now the next instruction fetch at clock cycle *i+4*, a glitch in this stage leads to a double-execution of `adds Rd,Rn`. This observation leads to the assumption that due to a glitch insertion in the fetch stage, the previous instructions remain in the fetch buffer and the current instructions are not executed at all. This assumption is confirmed by the fact that instead of the instruction `adds Rd,Rn` the `nop` instruction, fetched in cycle *i* is again present in the fetch buffer at the attacked cycle *i+2* and the addition is not executed at all. The same holds true for the attack at cycle *i+4* where `adds Rd,Rn`, fetched in cycle *i+2* is again in the instruction fetch buffer at cycle *i+4* and therefore executed twice.

Execution stage. Another phenomenon occurs if the clock glitch is inserted at cycle *i+5*, where the execution stage of the instruction `adds Rd,Rn` is attacked. In this case the value of register `Rd`, which stores the result of the addition, is set to a variation of wrong numbers. The resulting erroneous calculation results vary depending on the glitch period T_{Glitch} between 8.3 ns and 9.2 ns as well as the addends provided by the initialization values of Rd and Rn. This behaviour leads to the assumption, that a timing violation during the addition happens in case of attacking the execution stage of `adds`.

For a more detailed evaluation of the obtained results when attacking a single instruction, we further considered a consecutive execution of `adds` instructions as illustrated in Table IV, leading to the following observations:

TABLE IV. ATTACK ON A SET OF `ADDS RD,#imm` AND `ADDS RE,#imm` INSTRUCTION ON THE ARM CORTEX-M0 MICRCONTROLLER.

Instruction	**Cycle**									
	i	***i+1***	***i+2***	***i+3***	***i+4***	***i+5***	***i+6***	***i+7***	***i+8***	***i+9***
`adds Rd,#1`	F	D	E							
`adds Rd,#4`	F		D	E						
`adds Rd,#16`			F	D	E					
`adds Rd,#64`			F		D	E				
`adds Re,#1`					F	D	E			
`adds Re,#4`					F		D	E		
`adds Re,#16`							F	D	E	
`adds Re,#64`							F		D	E

Fetch state. When attacking the fetch stage in cycle *i+4*, the fetch operations of the two instructions `adds Rd,#16` and `adds Rd,#64` are affected simultaneously. Consequently, the fetch buffer is not updated with the new instructions: `adds Re,#1` and `adds Re,#4`. In contrast, the instructions previously fetched in clock cycle *i+2* remain in the fetch buffer. As a result, `adds Rd,#16` and `adds Rd,#64` are executed again in cycle *i+6* and in cycle *i+7*, respectively. The erroneous results when attacking the fetch stage of the ARM Cortex-M0 are determined by four instructions, since two instructions are executed twice, whereas the other two instructions are skipped.

Execution stage. In contrast to the fetch stage, the execution stage can be attacked in every clock cycle. This implies that attacking the fetch stage in cycle *i+4* comprises an attack on the execution of `adds Rd,#16` fetched in cycle *i+2*. However, a glitch in cycle *i+4* either affects the execution stage for a glitch period T_{Glitch} between 8.3 ns and 9.2 ns or the fetch stage for T_{Glitch} between 9.3 ns and 12.6 ns, whereas a clock glitch in cycle *i+3* only affects the execution of `adds Rd,#4`. Thus, a distinct selection of the affected pipeline stage is possible with properly chosen values for the clock glitch period T_{Glitch}.

When attacking the fetch stage for the other two investigated *arithmetical/logical instructions*, `muls Rd,Rn` and `lsls Rd,#imm` as listed in Table I, we obtained similar results compared to the `adds` instruction. However, when inserting a clock glitch in the execution stage, the effects are different depending on the attacked instruction. In case of attacking the execution of `lsls Rd,#imm` we observed that the register `Rd` is set to zero for a glitch period T_{Glitch} between 8.3 ns and 10.7 ns. With `muls Rd,Rn` the register `Rd` is set to a variation of wrong numbers for a glitch period T_{Glitch} between 10.2 ns and 20.7 ns. The erroneous value of the result register `Rd` increases with a higher value for T_{Glitch} and varies depending on the used initialization values for the operands. These relations between the erroneous result, the time range of T_{Glitch} and the attacked instruction can be described through diverging, instruction-dependent impact of the injected fault on different combinational logic parts of the CPU. Additionally, different operand values have impact on the propagation delay of the combinational logic.

2) ATxmega 256: For the ATxmega 256 we started by attacking *arithmetical/ logical instructions* using again the attack scenario described in Section III-C. More precisely, we investigated the behaviour of a single `add Rd,Rn` instruction and a single `mul Rd,Rn`, as stated in Table I, where the `mul` instructions takes two execution cycles. Considering the pipeline structure of the ATxmega 256 depicted in Figure 6, different results are expectable by attacking either the fetch stage or the execution stage of an instruction.

Fetch stage. When a glitch in inserted in the fetch stage, the `add Rd,Rn` instruction is not executed. As with the ARM Cortex-M0 this behaviour can be explained by the assumption, that instead of updating the instruction fetch buffer with `add Rd,Rn` the previous instruction, in our test scenario a `nop`, remains in the buffer. In particular, a glitch period T_{Glitch} between 5.9 ns and 17.9 ns led to this behaviour. Verified by an attack on the next fetch stage, where in normal conditions a `nop` should be fetched, the `add Rd,Rn` instruction remains in the fetch buffer and is subsequently executed twice. The only difference is given by fact that another range for T_{Glitch} between 9.3 ns and 15.3 ns is required in order to prevent the fetch buffer from being updated with the `nop` instruction. The reason for this behaviour can be explained by the assumption that different values of the fetch buffer result in different timing behaviour concerning the buffer update. Considering now the `mul Rd,Rn` instruction, we were able to achieve the same results with the same T_{Glitch} range when attacking the fetch stage of the instruction. In contrast to that, an attack on the next fetch stage, resulting in a double-execution for the `add` instruction, remained without any effects for the `mul` instruction.

Execution stage. When attacking the execution stage of the `add Rd,Rn` instruction with a glitch period T_{Glitch} between 5.9 ns and 7.2 ns, the value of the destination register `Rd` was changed to a constant value regardless of T_{Glitch}. Relations between other register values or memory entries were not observed during the experiments. Considering the `mul Rd,Rn` instruction, an attack on the second execution cycle led to wrong multiplication results using a glitch period T_{Glitch} between 6.4 ns and 7.8 ns. Erroneous values were only observed for the high byte of the 16-bit multiplication result when attacking the second execution cycle. However, no effects were observed when attacking the first execution cycle of the `mul` instruction.

3) Discussion/Comparison: Attacks on arithmetical/logical instructions led to similar behaviour on both MCU platforms. Due to the parallel fetch stage of the ARM Cortex-M0, two instruction fetches can be influenced with a single clock glitch. Furthermore, the T_{Glitch} values causing erroneous behaviour vary for the two platforms.

Skipping an instruction allows an attacker to output an internal state before a last modification, e.g. the AES state before the last key addition. With a pair of ciphertexts, once without and once with skipped last key addition, it is possible to calculate bytes of the last round key. Repeating one instruction can render several operations ineffective. If e.g., an internal value a is XOR'ed with a key byte k twice, this results in the original value a again ($a \oplus k \oplus k = a$). Setting the result of an arithmetical operation to a constant, known value (when attacking the execution stage) also poses a serious threat for attacks on cryptographic primitives implemented in software.

D. Branch Instruction

The second type of instructions investigated in our experiments are *branch instructions*. In contrast to the *arithmetical/logical instructions* these instructions modify the program flow without influencing data. Therefore, we were able to inject faults in the fetch stage preventing the overall execution process of a branch.

1) ARM Cortex-M0: For the ARM Cortex-M0 we attacked the `beq label` instruction directly located after a `cmp Ra,Rb` instruction in our test program. By inserting a glitch in the fetch stage of the branch instruction we obtained similar behaviour compared to the attack results for the fetch stage of the `adds Rd,Rn` instruction. When attacking the fetch stage of `beq label` the branch is prevented from being executed, although the corresponding registers, used by the compare instruction, are initialized with the same values in order to fulfill the condition for taking the branch. This behavior can be achieved between 9.3 ns and 12.6 ns for the clock glitch period T_{Glitch}.

2) ATxmega 256: In case of attacking the `breq label` instruction of the ATxmega 256, the branch instruction is executed directly after the compare instruction `cp Ra,Rb`. When attacking the fetch stage of the branch instruction with a glitch period T_{Glitch} between 6.7 ns and 18.2 ns the branch is not taken, although the branch condition is fulfilled using the previously executed compare instruction. Based on the obtained results when attacking the fetch stage of *arithmetical/logical instructions*, it can be assumed that the previous compare instruction is executed twice instead of `breq label`.

3) Discussion/Comparison: With similar effects, both MCU platforms were vulnerable to attacks in the fetch stage of *branch instructions*. The only difference when obtaining this behaviour between the ARM Cortex-M0 and the ATxmega 256 was the glitch period T_{Glitch}.

Preventing a branch from being executed enables, for example the option of skipping security-relevant code segments, loop iterations or complete function calls. In this context, a manipulation of algorithms implemented in software dealing with sensitive data might lead to a leakage of valuable or secret information processed on an MCU. A particularly interesting fact is, that branches are usually executed directly after a compare instruction. Assuming that the compare instruction remains in the instruction fetch buffer, the compare instruction is executed twice instead of the branch instruction. Thus, it is possible to change the program flow without any undesirable side effects on data or other parts of the code.

E. Memory Instructions

To analyse the resulting effects of clock glitches on the group of *memory instructions* we considered load and store instructions for both microcontrollers, the ARM Cortex-M0 and the ATxmega 256 as listed in Table I. Again, either attacking the fetch stage or the execution stage of an instruction execution process led to different results.

1) ARM Cortex-M0: The instructions `ldr Rd,[Rn]` and `str Rd,[Rn]` were investigated for the ARM Cortex-M0. Register `Rn` holds the SRAM address for either storing or loading the data value. Both instructions require two execution cycles.

Fetch stage. When inserting a glitch with T_{Glitch} between 10.2 ns and 10.9 ns in the fetch stage of `ldr Rd,[Rn]` the loaded value was zero instead of the value stored at the source address. In case of the `str Rd,[Rn]` instruction the value stored was zero instead of the initial value of the `Rd`. These are two particularly interesting facts, as a different range for T_{Glitch} is required compared to the *arithmetical/logical instructions* and *branch instructions*. Additionally, the behaviour of loading zero when attacking `ldr Rd,[Rn]` or storing zero when attacking `str Rd,[Rn]` is different compared to all previous experiments concerning the fetch stage.

Execution stage. When attacking the execution of either `ldr Rd,[Rn]` or `str Rd,[Rn]`, our results show that only the second execution cycle is vulnerable to clock glitch insertion. With a glitch period T_{Glitch} between 8.3 ns and 8.9 ns we observed that the SRAM address stored in register `Rn` is either loaded into `Rd` when attacking `ldr Rd,[Rn]` or stored to the SRAM address when attacking `str Rd,[Rn]`. In case of `ldr` the corresponding behavior equals a register transfer instruction, i.e. `mov Rd,Rn`. For values of T_{Glitch} between 9.0 ns and 9.6 ns, regardless of which memory instruction is attacked, neither `ldr Rd,[Rn]` nor `str Rd,[Rn]` is executed. Thus, it is possible to prevent memory instructions from being executed.

2) ATxmega 256: For the ATxmega 256 we target on `ld Rd,X` and `st X,Rn` where `X` holds the 16 bit SRAM address as a combination of register `R27` and `R28`. It is notable that the `ld` instruction requires two execution cycles, whereas the execution of the `st` instruction is performed within one clock cycle.

Fetch stage. The instructions `ld Rd,X` and `st X,Rn` were not executed when using a glitch period T_{Glitch} between 5.9 ns and 17.9 ns for an attack in the fetch stage. This behaviour can be explained due to the fact that the injected fault prevents the instruction fetch buffer from being updated as we already observed in our previous experiments concerning the fetch stage.

Execution stage. When `ld Rd,X` is attacked in the first execution cycle wrong data is written to the destination register `Rd`. In this context, no stable relations between the resulting wrong value of `Rd` and any value at a different memory location are given. The second execution cycle of `ld Rd,X` exposed to be resistant against clock glitch attacks. When attacking `st X,Rn` in its execution stage wrong data is written to the memory address, which is defined by the address pointer `X`. For both instructions, the erroneous behaviour is achieved when using a clock glitch period T_{Glitch} between 8.1 ns and 8.8 ns. Increasing T_{Glitch} to a value between 9.6 ns and 10.3 ns, register `Rd` is set to zero in case of attacking `ld Rd,X` in the first execution cycle, while for `st X,Rn` no effects were observed.

3) Discussion/Comparison: Injected faults led for both MCU platforms to either loading constant values or to preventing a load or store instruction from being executed. Additionally, constant values, namely zero or the memory address are stored instead of the desired register entry in case of the ARM Cortex-M0.

In this context, loading a known and constant value instead of true memory entries enables an attack scenario on algorithms which are based on retrieving secret information from memory. These attacks present a threat to cryptographic implementations where memory instructions are for example used to load keys or initial values for random number generators. Again, influencing these values before they are applied to cryptographic primitives might establish an access to sensitive data.

V. Discussion with Related Work

In this section a comparison of the results achieved during this work with previous work is done. We limit the comparison to the work of Balasch et al. [3]. The experiments in their work are quite similar to the ones we have performed and also the investigated microcontrollers are comparable.

The authors in [3] have used five samples of the same microcontroller in order to verify the scattering of the parameters required for inducing similar faults. They mention that there are small variations from one sample to the next one. In contrast to that, we have only used one sample per platform, but we have evaluated two different platforms. For each fault type, we give an interval of the parameter, in which the fault is induced with 100% probability. We are confident that for different samples of the same microcontrollers, these intervals will overlap resulting in parameters valid for a wide range of samples minimizing the impact due to process variations.

For attacks targeting the fetch-stage, a subset of the results obtained in [3] are comparable with our results. For a specific glitch interval, the previous instruction remains in the fetch buffer instead of loading the current one. The current instruction is not executed as a matter of fact. In [3] the authors mention, that they were also able to modify the opcode of the current instruction using different values for the glitch length. This behaviour could not be achieved with the microcontroller platforms investigated in this work.

Clock glitches inserted during the execution of an instruction also lead to erroneous behaviour on the investigated microcontroller in [3]. The fault injection leads to wrong calculation results. The observable fault depends on several factors: values of the operands, used registers, targeted instruction as well as shape of the clock glitch. Similar behaviour can also be observed for both of our investigated microcontroller platforms.

The authors in [3] also face the same problem as we do: Due to the pipeline structure for the instruction processing (*fetch* of the next instruction and *execute* of the current instruction happen in the same clock cycle), two instructions are affected by inserting a single clock glitch. Our experiments revealed, that depending on the shape of the clock glitch (T_{Glitch} setting) either the fetch or the execute stage can be attacked without affecting the other stage. Figure 8 can assist in finding such special T_{Glitch} settings. E.g., if an attacker only wants to influence the fetch stage of a specific multiplication on the ARM Cortex-M0, she should use a T_{Glitch} value of about 10 ns. If, on the other hand, only the execution stage of a multiplication should be affected, a T_{Glitch} value of 14 ns can be used.

VI. Conclusion

In this work, the effects of practical fault attacks using clock glitches on two different microcontrollers (ARM Cortex-M0 and ATxmega 256) are discussed in detail. Results show that the *fetch* stage and the *execute* stage of the CPU pipeline are affected by clock glitches and different instructions lead to different behaviour. None of our experiments targeting the ARM Cortex-M0 MCU allowed inducing faults during the *decode* stage of the instruction pipeline. Furthermore, a new approach taking advantage of short-time underpowering in addition to clock glitches is presented, which allows to significantly improve the fault attacks in terms of fault reproducibility. In the result section we give intervals for the parameters influencing the clock-glitch shape, in which the fault occurrence probability equals 100 %. We only evaluated one sample per microcontroller platform, so we did not analyze the variation of the parameters due to process variations. But we are confident that the intervals we give allow to compensate for the sample variation.

Both investigated microcontrollers are frequently applied in sensor nodes, which makes them a potential target for physical attacks. Our results show that software implementations handling sensitive data on this microcontrollers need to be protected against this kind of attacks with great care. Discussions about potential attacks taking advantage of the uncovered vulnerabilities are done in the result section. The results of this work should assist software developers to identify and understand the threats, which arise due to fault attacks. As a result, appropriate countermeasures can be integrated.

Acknowledgment

This work has been supported by the European Commission through the FP7 program under project number 610436 (project MATTHEW).

References

[1] M. Agoyan, J.-M. Dutertre, D. Naccache, B. Robisson, and A. Tria. When Clocks Fail: On Critical Paths and Clock Faults. In *Smart Card Research and Advanced Application*, pages 182–193. Springer, 2010.

[2] Atmel Corporation. 8/16-bit Atmel XMEGA A3U Microcontrollers. Available online at http://www.atmel.com/Images/Atmel-8386-8-and-16-bit-AVR-Microcontroller-ATxmega64A3U-128A3U-192A3U-256A3U_datasheet.pdf, 2013.

[3] 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 *FDTC, 2011*, pages 105–114. IEEE, 2011.

[4] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. Cryptology ePrint Archive (http://eprint.iacr.org/), Report 2004/100, 2004.

[5] A. Barenghi, G. M. Bertoni, L. Breveglieri, and G. Pelosi. A Fault Induction Technique Based on Voltage Underfeeding with Application to Attacks Against AES and RSA. *Journal of Systems and Software*, 86(7):1864–1878, 2013.

[6] E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In B. S. K. Jr., editor, *CRYPTO '97, Proceedings*, volume 1294 of *LNCS*, pages 513–525. Springer, 1997.

[7] J. Blömer and J.-P. Seifert. Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In R. N. Wright, editor, *Financial Cryptography, FC 2003*, volume 2742 of *LNCS*, pages 162–181. Springer, January 2003.

[8] H. Choukri and M. Tunstall. Round Reduction Using Faults. *FDTC*, 5:13–24, 2005.

[9] A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria. Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES. In *FDTC, 2012*, pages 7–15, Sept 2012.

[10] A. Dehbaoui, A.-P. Mirbaha, N. Moro, J.-M. Dutertre, and A. Tria. Electromagnetic Glitch on the AES Round Counter. In *COSADE, 2013*, pages 17–31. Springer, 2013.

[11] S. Endo, T. Sugawara, N. Homma, T. Aoki, and A. Satoh. An on-chip glitchy-clock generator and its applicataion to safe-error attack. In *COSADE, 2011*, Workshop Proceedings COSADE 2011, pages 175–182, 2011.

[12] T. Fukunaga and J. Takahashi. Practical Fault Attack on a Cryptographic LSI with ISO/IEC 18033-3 Block Ciphers. In *FDTC, 2009*, pages 84–92. IEEE, 2009.

[13] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *CHES, 2001, Proceedings*, volume 2162 of *LNCS*, pages 251–261. Springer, 2001.

[14] L. Hemme. A Differential Fault Attack Against Early Rounds of (Triple-) DES. In *CHES, 2004*, pages 254–267. Springer, 2004.

[15] M. Hutter and J.-M. Schmidt. The Temperature Side-Channel and Heating Fault Attacks. In *CARDIS 2013*, LNCS, 2013. in press.

[16] C. H. Kim and J.-J. Quisquater. Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures. In *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, pages 215–228. Springer, 2007.

[17] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, pages 104–113, 1996.

[18] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO*, pages 388–397, 1999.

[19] O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-resistant Smartcard Processors. In *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, pages 2–2. USENIX Association, 1999.

[20] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *FDTC, 2013*, pages 77–88. IEEE, 2013.

[21] NXP. LPC1110/11/12/13/14/15 Product Data Sheet. Available online at http://www.nxp.com/documents/data_sheet/LPC111X.pdf, December 2013.

[22] J.-J. Quisquater and D. Samyde. Eddy Current for Magnetic Analysis with Active Sensor. In *E-Smart, 2002*, pages 185–194. UCL, September 2002.

[23] J.-M. Schmidt and C. Herbst. A Practical Fault Attack on Square and Multiply. In *FDTC, 2008*, pages 53–58. IEEE, 2008.

[24] N. Selmane, S. Guilley, and J.-L. Danger. Practical Setup Time Violation Attacks on AES. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, pages 91–96. IEEE, 2008.

[25] S. Skorobogatov. Low temperature data remanence in static RAM. Technical report, University of Cambridge Computer Laboratory, June 2002.

[26] S. P. Skorobogatov. *Semi-invasive attacks - A new approach to hardware security analysis*. PhD thesis, University of Cambridge - Computer Laboratory, 2005. Available online at http://www.cl.cam.ac.uk/TechReports/.

[27] S. P. Skorobogatov and R. J. Anderson. Optical Fault Induction Attacks. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 2–12. Springer, 2003.

[28] E. Trichina. Multi-Fault Laser Attacks on Protected CRT RSA. Invited Talk - FDTC 2010, 2010.

[29] L. Zussa, J.-M. Dutertre, J. Clediere, and A. Tria. Power Supply Glitch Induced Faults on FPGA: An In-depth Analysis of the Injection Mechanism. In *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, pages 110–115. IEEE, 2013.