Investigating the Vulnerabilities of Two Microcontroller Platforms to Fault Injection Attacks

Michael Höfler
michael.hoefler@student.tugraz.at

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master Thesis

Supervisor: Dipl.-Ing. Thomas Korak
Assessor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Karl-Christian Posch

May, 2014

I hereby certify that the work presented in this thesis is my own work and that to the best of my knowledge it is original except where indicated by reference to other authors.

Ich bestätige hiermit, diese Arbeit selbständig verfasst zu haben. Teile der Diplomarbeit, die auf Arbeiten anderer Autoren beruhen, sind durch Angabe der entsprechenden Referenz gekennzeichnet.

Michael Höfler

Acknowledgements

I would like to express my greatest appreciation to my supervisor, Thomas Korak, who provided me with exceptional guidance. His advices and insightful comments have been a great and constructive help in all stages of my thesis.

I would also like to extend my sincerest thanks to the people at the *Institute for Applied Information Processing and Communications* who were always willing to provide help and assistance.

I would like to thank my entire family for always supporting me and bringing me where I am today. I owe my deepest gratitude to my girlfriend, Miri, for her exceptional patience during my work.

Finally, I would like to thank all my friends for their wonderful company and the great time we had together during my years of study in Graz. Last but not least, special thanks to Hanna Blum for proofreading my thesis.

Abstract

Cryptographic algorithms are widely used to provide protection for confidential information against unauthorized access. The underlying cryptographic primitives are proven to be mathematically secure. However, implementations of such primitives for specific applications can still leak some valuable information through side channels or by introducing faulty behavior. Fault attacks particularly exploit these implementation-specific weaknesses by actively influencing the behavior of a device in order to produce computational errors and, furthermore, to reveal secret information. The strategies for influencing the behavior in order to induce faults are based on changing environmental operating conditions and vary depending on the applicable effort for an attack. Typical attack strategies are, for instance, tampering with the clock signal or supply voltage level, influencing the integrated circuit with focused laser beams or establishing electrical contact to the interconnect lines. Particularly, due to the huge number of different available fault injection mechanisms, considering any potential threat for securing a device against fault attacks has become a challenging task. The potential threat of fault attacks has to be considered particularly in scenarios, where an adversary is able to gain physical access to the device. Sensor networks, smart-card systems or, more generally, mobile devices are among such attack scenarios. Typically, microcontrollers are used in these devices to implement a variety of different tasks, including cryptographic operations, like authentication or data encryption.

This thesis presents an in-depth analysis of the vulnerability of two different microcontroller platforms on fault attacks using clock glitch insertion and supply voltage manipulation. Both strategies aim at inducing faults due to a timing-constraint violation of the logic blocks. To reveal insight into the occurring effects, the influence of clock glitch attacks on the instruction processing-sequence of several assembly instructions is investigated for both microcontrollers. Furthermore, we present a novel approach of combining short-time underpowering with clock-glitch insertion to increase the sensitivity of the device to fault injection. Results of practically performed experiments on both micocontrollers show that the most reliable faults can be induced when attacking the fetch stage of the instruction pipeline. In this case, fetching the new instruction is prevented and the previous instruction remains in the fetch buffer. Different effects are observed when attacking the execution stage of the instruction pipeline where an induced fault mainly results in erroneous calculations. Finally, the effects of clock glitch attacks on the built-in hardware support for the Advanced Encryption Standard (AES), provided by one of the two investigated microcontrollers, are analyzed. Therefore, we present a fault-based black-box characterization for retrieving detailed information about the underlying hardware structure. Derived from the obtained results when attacking the AES encryption procedure, we introduce two key-retrieval attacks using only faulty ciphertexts.

The presented results and the analysis of the effects of fault attacks on the investigated microcontrollers reveal detailed information about the potential risk of these kind of attacks on specific cryptographic primitives and implementations.

Keywords: fault attack, clock glitch, microcontroller, AES

Kurzfassung

Kryptographische Algorithmen werden in vielen Bereichen eingesetzt, um vertrauliche Information gegen unautorisierten Zugriff zu schützen. Die dafür zugrunde liegenden kryptographischen Primitive gelten nachweislich als mathematisch sicher. Dennoch ist es möglich, dass anwendungsspezifische Implementierungen dieser Primitive wertvolle Informationen über Seitenkanäle preisgeben. Fehlerattacken versuchen Berechnungsfehler zu verursachen, indem sie das Verhalten eines Gerätes aktiv beeinflussen, um so geheime Information herauszufinden. Diese Fehler werden durch gezielte Veränderung von Betriebsbedingungen eines Gerätes erzeugt, wobei die dafür notwendigen Strategien von Manipulation des Taktsignals oder der Versorgungsspannung über die Beeinflussung von integrierten Schaltkreisen mit einem fokussiertem Laserstrahl bis hin zur Herstellung von elektrischem Kontakt zu den Verbindungsleitungen des Schaltkreises reichen. Insbesondere die breite Auswahl an unterschiedlichen Fehlermechanismen, erschwert die Anwendung eines zuverlässigen Schutzmechanismus. Typische Ziele von Fehlerangriffen sind Sensor-Netzwerke, Smartcard Systeme oder generell mobile Geräte, bei denen sich der Angreifer temporär Zugang zum Gerät verschaffen kann. In solchen Systemen kommen üblicherweise Mikrocontroller zum Einsatz, da diese eine Vielzahl von Aufgaben übernehmen können. Diese Aufgaben beinhalten, unter anderem, kryptographische Operationen wie etwa Authentifizierung und Datenverschlüsselung.

Diese Arbeit beschreibt eine detaillierte Analyse der Schwachstellen von zwei unterschiedlichen Mikrocontrollern bezüglich Fehlerattacken. Durch Einfügen von Störimpulsen im Taktsignal und mittels Manipulation der Versorgungsspannung wird gezielt fehlerhaftes Verhalten hervorgerufen. Beide Methoden verfolgen dabei den Ansatz einer Fehlergenerierung durch Verletzen der zeitlichen Kriterien von Logikblöcken. Um die auftretenden Effekte zu beschreiben, wird der Einfluss von Störimpulsen im Taktsignal auf die Ausführungssequenz von Assembler Instruktionen analysiert. In diesem Zusammenhang stellen wir einen neuartigen Ansatz vor, der kurzzeitiges Absenken der Versorgungsspannung mit dem Einbringen von Störimpulsen im Taktsignal kombiniert, wodurch eine Erhöhung der Fehlerempfindlichkeit des Gerätes erreicht wird. Die Ergebnisse der durchgeführten Experimente zeigen für beide Mikrocontroller, dass während des Ladevorgangs eines Befehls am zuverlässigsten Fehler generiert werden können. Dabei wird das Laden eines neuen Befehls verhindert und der zuvor geladen Befehl verbleibt im Befehlspuffer. Andere Effekte treten bei einem Angriff auf die Befehlsausführung auf, wobei in diesem Fall vorwiegend falsche Berechnungsergebnisse erzeugt werden. Abschließend werden die Auswirkungen von Störimpulsen im Taktsignal auf die AES (Advanced Encrytpion Standard) Hardwareeinheit von einem der beiden Mikrocontroller analysiert. In diesem Zusammenhang präsentieren wir eine fehlerbasierte Charakterisierung der Hardwarestruktur, sowie zwei davon abgeleitete Angriffe auf die AES Verschlüsselung. Beide Angriffe benötigen lediglich fehlerhafte Geheimtexte um den geheimen Schlüssel zu erhalten.

Die Ergebnisse der auftretenden Effekte von Fehlerangriffen auf Mikrocontrollern stellen detaillierte Information über das potentielle Risiko dieser Angriffe auf kryptographische Primitive und Implementierungen dar.

Stichwörter: Fehlerangriff, Taktsignal, Störimpuls, Mikrocontroller, AES

Contents

Chapter 1

Introduction

Protecting secret information against unauthorized access has become an important aspect in most kinds of modern devices nowadays. User data on smart phones, measurement data in sensing networks or communication data over wireless links are just a few examples. Depending on the field of application, different cryptographic algorithms and primitives can be applied in order to ensure confidentiality, integrity and authenticity (CIA) of data. When selecting cryptographic algorithms for devices dealing with sensitive data, the capabilities for an attacker who physically accesses such a device are often disregarded. These so-called physical attacks do not necessarily aim at mathematical weaknesses of algorithms or implementations but rather take advantage of physically observable or manipulable parameters. In case of passive attacks, leaking side-channel information during cryptographic calculations is observed and analyzed. Typical side-channel information is given by the power consumption [22], the electromagnetic emanation [17], or timing behavior of a device used to retrieve secret information. With accurate time measurement for example, it is possible to retrieve information about executed algorithms or processed data if the required execution time varies depending on the given input data. With this approach, Kocher [23] showed that it is possible to find the entire secret key of a cryptographic system. In contrast to the passive side-channel attacks, fault attacks actively affect a device and its operation by modifying physical/environmental parameters. Thus, it is possible to enforce faulty behavior or wrong conditions in the attacked device. As a consequence, induced faults typically result in erroneous calculations. This enables an attacker to gain details about the implementation or to even extract secret information. Fault Attacks are typically categorized into non-invasive, semi-invasive and invasive attacks. Covering the manipulation of physical parameters like the temperature [18], the supply voltage [43] or the clock signal [7] of a device, non-invasive attacks usually do not require conspicuous modifications of the attacked device. As a result, it is highly probable that an attack remains hidden. In contrast to non-invasive attacks, the group of semi-invasive and invasive attacks requires more sophisticated approaches. The device package has to be removed in order to get access to the inner semiconductor structure. By exposing the chip surface to intense white light or focused laser beams, faults can be induced due to photoelectric effects [27]. Skorobogatov and Anderson [37] presented an optical fault injection attack on a microcontroller successfully influencing the content of memory cells. Removing the device package in order to perform fault attacks has the advantage that fine-grain faults can be injected in terms of timing as well as affected values. Detailed categorizations of different fault attack methods including experimental approaches and a discussion of feasible countermeasures are given by Bar-El et al. [8].

In general, limitations of fault attacks are given by the fact that physical access to the device is required. However, concerning non-invasive attacks, an attacker only has to be able to control a device for a certain period of time. Since the target remains intact, the device can be returned after an attack inconspicuously without leaving visible traces. As a consequence, mobile applications represent an interesting target group for non-invasive fault attacks since it is easily possible to gain access for an attacker. Especially devices involved in cryptographic procedures like smart cards, radio-frequency identification (RFID) technologies and wireless sensing platforms have to be considered as potential targets. In a wide range of these applications microcontrollers are used to perform several tasks, including security-relevant calculations. Additionally, hardware-assisted cryptographic features intensify the use of microcontrollers for applications dealing with sensitive data. The features allow to speed-up cryptographic calculations on the one hand and decrease the power consumption compared to a pure software implementation on the other hand.

This work analyzes the vulnerability of microcontrollers to non-invasive fault attacks in a practical approach. Two different microcontroller platforms (Atmel ATxmega 256 and ARM Cortex-M0) are evaluated. For both microcontroller units (MCU), the focus is put on a detailed analysis of how injected faults affect the instruction execution process. Additionally, the AES (Advanced Encryption Standard) crypto engine of the ATxmega 256 is investigated in terms of vulnerability to fault attacks. Therefore, it is shown how injected faults can be used to gain information about the underlying hardware structure of the implementation of a cryptographic algorithm as well as how it is possible to retrieve the secret key used for encryption. The presented results should serve as a basis for investigating the potential risk of fault attacks on specific cryptographic primitives and implementations.

1.1 Contribution

In this work we present the practical approach of injecting faults into two different microcontroller platforms by applying non-invasive fault attacks. More precisely, the vulnerabilities to clock glitch attacks targeting a selection of comparable instructions for the Atmel ATxmega 256 and the ARM Cortex-M0 are analyzed. In this way, a direct comparison of two different MCU architectures to similar fault injection is given. In addition to a fault analysis of the instruction execution procedure, we present two key retrieval attacks on the AES crypto engine of the ATxmega 256 by applying clock glitches during the encryption procedure. In detail, this work provides the following contributions:

- Clock glitch attacks aim at violating the timing constraints of a device by applying a manipulated clock signal beyond the specified ratings. As a consequence, the instruction execution procedure of a microcontroller can be influenced. Results show, that inducing faults with accurate timing leads to a controlled manipulation on both investigated microcontroller. In due consideration of the different MCU architectures, we are able to compare the effects of similar faults on the two-stage pipeline of the ATxmega 256 and the three-stage pipeline of the Cortex-M0. To provide a reliable comparison of both MCU platforms, the effects of clock glitches on three different groups of instructions are evaluated: *arithmetical/logical instructions*, *branch instructions*, and *memory instructions*.
- We introduce an approach of combining clock glitch attacks and supply voltage attacks, where short-time underpowering of the device is simultaneously performed

with a manipulation of the clock signal. The voltage level used for underpowering is chosen in a way that only applying short-time underpowering does not affect the device at all. The effective injection of faults is only done by simultaneously inserting clock glitches to ensure accurate timing and reproducibility. Results show that the efficiency of clock glitch attacks can be increased significantly by applying this approach on the Cortex-M0.

- The Cortex-M0 provides a brown-out detection for monitoring the power supply. This feature can serve as a simple countermeasure against underpowering attacks. Therefore, we evaluate the brown-out detection by analyzing its response to supply voltage reduction for different durations and voltage levels. By choosing appropriate values for voltage and duration, the brown-out detection mechanism fails to detect an underpowering attack.

- We further analyze how the AES crypto engine of the ATxmega 256 is influenced by clock glitch attacks. Consecutively inducing single clock glitches during AES encryption makes it possible to retrieve information about the hardware implementation. Depending on the attacked cycle of encryption, different erroneous ciphertexts are observed, allowing to draw conclusions about the underlying hardware structure. We present details about this approach of fault-based black-box characterization for the AES crypto engine of the ATxmega 256 as well as the obtained results.

- Additionally, we present two key-retrieval attacks on the AES crypto engine of the ATxmega 256 by injecting faults during AES encryption. By inserting clock glitches during the final encryption round of the AES, an analysis of resulting erroneous ciphertexts enables us to define two fault models for key-retrieval attacks. For both attacks it is possible to reveal the secret key used for encryption only by observing faulty ciphertexts.

1.2 Outline

The outline of this work is as follows: First, we discuss relevant related work in Chapter 2, focusing on fault attacks performed on different applications and implementations. Next, a general categorization and description of fault attacks is given in Chapter 3 including a comprehensive discussion of how clock and supply voltage manipulation is used in order to perform fault attacks. In Chapter 4 we introduce the fault injection setup and the microcontrollers used for our experiments. A description of the attack scenarios and the investigated instructions is given in Chapter 5 followed by the achieved results for the Cortex-M0 and the ATxmega 256 in Chapter 6. Here we present common and individual characteristics for both microcontrollers, obtained during our experiments. In Chapter 7 a general introduction to the Advanced Encryption Standard (AES) is given as well as a description of the attack scenario for the AES crypto engine of the ATxmega 256. Derived from the obtained results, the implementation analysis of the AES crypto engine and the applied key retrieval attacks are explained in Chapter 8. Conclusions are drawn in Chapter 9.

Chapter 2

Related Work

In recent years, considering fault attacks as potential threats to cryptographic applications has become increasingly important. By now, a wide range of literature and publications address these attacks covering hardware implementations as well as software implementations of cryptographic primitives. The focus of research for the category of non-invasive fault attacks varies from theoretical analysis of specific cryptographic algorithms to implementation attacks performed on devices in practice. Fault attacks can be categorized depending on the intended impact on a device and which faulty behavior can be exploited. Therefore, *differential fault analysis (DFA)*, first introduced by Biham and Shamir [9], aims at retrieving secret information by comparing correct and fault-induced results of a cryptographic calculation. Yen and Joye [41] propose the concept of *safe-error attacks* analyzing whether an injected fault leads to a erroneous calculation or not. The third category of fault attacks are *algorithm modifications* where specific manipulations of cryptographic calculations are exploited to retrieve secret information.

Djellid-Ouar et al. [13] discuss the effects of supply voltage glitches on complementary metal-oxide-semiconductor (CMOS) circuits based on a detailed analysis of semiconductor characteristics. Therefore, the effects of glitches influencing the behavior of D-flip-flops on the one hand and combinational logic on the other hand are investigated and verified by applying circuit simulations. An early work of Koemmerling and Kuhn [24] covers a wide range of physical attack techniques on microcontrollers for extracting secret data. Additionally, they give an overview about hardware countermeasures to prevent an attack or at least to increases an attacker's effort in order to succeed. Fuhr et al. [15] propose a theoretical discourse of several fault attacks on the AES algorithm. Their approach relies on non-uniform fault distribution models where only a set of faulty ciphertexts is necessary to recover the secret key.

In addition to the aforementioned works, a huge number of reports present practically applied attacks. Li et al. [26] discuss the impact of non-uniform distributed faults on a hardware implementation of AES. Focusing on the circuit structure of the S-Box, they state the reasons for the non-uniformity of faulty S-Box outputs and show that specific values appear more likely under faulty conditions. By applying clock glitch attacks and using hardware simulation, their theoretical assumptions could be verified. On this basis they additionally presented an attack using electromagnetic interference in which they successfully recovered several key bytes. Without using trigger signals for accurate fault injection timing their attack scenario can be considered as very realistic. By tampering with the supply voltage, Selmane et al. [35] describe a practical attack on AES co-processor of a smart card. Based on underpowering of the device during the whole encryption

process, setup time violations on the critical path of combinational logic lead to fault injection. Beside the effect of multiple errors caused by lowering the supply voltage, they were also able to induce single faults. An FPGA-based AES implementation was attacked by Agoyn et al. [1] by applying clock glitches in order to induce faults. Practical clock glitch attacks were used to characterize the injected faults and to verify their theoretical analysis. Fukunaga and Takahashi [16] present practical fault attacks on a cryptographic application-specific integrated circuit (ASIC). They developed an experimental setup for injecting faults into a desired cycle of the six symmetric block ciphers implemented on the ASIC to analyze the vulnerability on clock glitch attacks.

An attack based on manipulating the round counter of a round-based encryption algorithm is shown by Choukri and Tunstall [10]. Therefore, an AES implementation running on a microcontroller-based smart card was attacked using supply voltage glitches to reduce the AES execution to only one round. Their work also provides information about how the smart card is forced to faulty behavior in order to change the program flow of the AES software implementation. Dehbaoui et al. [12] present another attack on manipulating the round counter of an AES software implementation using electromagnetic pulses for fault injection. They were able to prevent an incrementation of the round counter and to redundantly execute an encryption round. Based on this approach, they describe a round addition attack for recovering the secret key.

Schmidt and Herbst [33] performed a practical fault attack on the square and multiply algorithm of a Rivest Shamir Adleman (RSA) implementation using supply voltage spikes. In doing so, they were able to manipulate the program flow of the attacked microcontroller implementation leading to skipped square operations further used to recover the secret key. Focusing on the Chinese Remainder Theorem (CRT), Kim and Quisquater [20] present another attack on an RSA implementation. By tampering with the power supply voltage, they apply double-fault injection in order to first influence cryptographic computations and second, to skip a fault detection routine. Due to this weakness, commonly used countermeasures for secure RSA implementations can be bypassed.

So far, all practical attacks mentioned, target a specific cryptographic hardware or software implementation. In contrast to these works, Moro et al. [28] present another approach by not focusing on a specific implementation but rather analyzing the overall behavior of an ARM Cortex-M3 microcontroller to electromagnetic fault injection. Based on their experimental results, they present a fault characterization and analysis describing how different attack parameters influence the microcontroller. Balasch et al. [7] use a similar approach by applying clock glitch attacks on an Atmel ATmega 162 microcontroller and investigating the effects of injected faults on the instruction set. Based on several attacks on different instructions, they were able to show that varying timing parameters for the clock glitch can lead to different manipulations on the executed instruction. In particular, they analyze the effects on the program flow or the data flow and which parts of the MCU are influenced by their attacks.

In all conscience there is only a small number of reports taking a similar approach to those of Balasch et al. [7] and Moro et al. [28]. Similar to [7], this work relies on investigating the impact of non-invasive fault injection attacks on two different MCU platforms. In addition to [7], analyzing two MCU types on the impact of clock glitch attacks allows us to specify relationships and give detailed comparisons based on the observed results. This approach facilitates a generalizable characterization of the fault injection effects on microcontrollers. Additionally, we present an implementation-specific attack on the AES crypto engine of the ATxmega 256. Compared to the aforementioned

practical attacks, we applied a black-box characterization of the AES implementation based on faulty ciphertexts, to derive possible fault models. Based on the observed results, we present two key retrieval attacks. A detailed contribution to this work is given in Section 1.1.

Chapter 3

Fault Injection Methodology

Typically, the access to processed data of a cryptographic device is severely restricted, meaning that only specific input and output data is available to an adversary. Intermediate values during computation can be used to break a secure system. Fault attacks aim at revealing such secret or obscured information by actively influencing a device in its operation. The idea behind fault attacks relies on selectively manipulating or modifying the behavior of a device in order to induce faults during computations. Therefore, the strategies for manipulating the behavior of a device range from violating operation conditions by taking influence on environmental parameters to penetration for physically accessing the inner structure of the device. Regardless of the applied strategy, a restrictive condition for performing active fault attacks is given by the fact that physical access to the device is necessary. However, fault attacks represent a potential threat for cryptographic devices. By analyzing the impact of the induced faults within the computation result, it is possible to obtain secret information or implementation details. Fault attacks usually exploit weak points of a specific implementation design and the underlying hardware structure of a cryptographic primitive.

Due to a huge number of different applicable fault injection strategies, securing a device against active fault injection is rather difficult. Furthermore, additional implementation overhead and hardware design constraints might lead to a reduced application of proper countermeasures against fault attacks. In this context, Karaklajić et al. [19] present an intuitive guide for overcoming the complexity of considering countermeasures during hardware design.

In this chapter we introduce common fault injection strategies by a general description of three different groups of active fault attacks: *invasive attacks*, *semi-invasive attacks*, and *non-invasive attacks*. For each category general properties and the required effort for performing an attack are presented. Following, we particularly focus on non-invasive attack approaches used in our practical experiments for investigating the vulnerability of microcontrollers to fault injection. Clock glitch attacks and supply voltage manipulation are applied in the practical experiments, presented in this work. Furthermore, we explain the underlying attack mechanism and how clock glitches and underpowering can influence the hardware circuit based on timing constraints violations.

3.1 Non-Invasive Fault Attacks

The group of non-invasive fault attacks are considered to be the simplest and certainly cheapest form of active fault injection. In this case, faults are, for example, induced by tampering with environmental conditions or supply signals without requiring sophisticated modifications of the attacked device. Therefore, a rather cheap and simple equipment is sufficient in order to perform these kind of attacks. The most important restrictions of non-invasive fault attacks are given by the fact, that usually the entire device or several parts of the device are influenced at once. Thus, focusing fault injection on a specific part of the device is rather difficult which complicates a reliable localization of the induced faults necessary to identify the affected part on the device. The most common non-invasive fault attacks are based on the following fault injection strategies.

- **Clock glitches.** Variations in the clock signal can be applied to devices that require an external clock source. Adding additional clock edges with a shorter period than supported by the device might cause, for instance, misinterpreted assembly instructions executed on a microcontroller [7]. Therefore, clock glitch attacks aim at manipulating the timing behavior of a hardware implementation by violating the critical path delay of the combinational logic.

- **Power supply manipulation.** The approach of tampering with the power supply voltage of devices again utilizes the manipulation of timing behavior similar to clock glitch attacks. Precisely induced negative or positive spikes in the power supply line can, for instance, be applied to a microcontroller in order to modify the program flow [10] or used to influence the operation of an AES hardware implementation on a field-programmable gate array (FPGA) [43]. In contrast to these attacks, underpowering represents another approach for power supply manipulation. The supply voltage is reduced below the specified operating range of a device for a certain period of time until faults can be recognized.

- **Temperature.** Fault induction can be achieved by changing the temperature of an electronic device to a very high or low temperature beyond the specified operating range for correct operation. Heating up a microcontroller above 150 °C can induce faults during operation [18]. In this context, Hutter and Schmidt [18] additionally present data remanence attacks in which overheating is used to stress the internal static random-access memory (SRAM) of a microcontroller in order to provoke a stable power-up state of memory cells.

- **Electromatgnetic pulses.** High-frequency electromagnetic fields induce eddy currents in the chip which in turn can influence data signals. By inducing a high voltage into an electromagnetic probe, it is possible to generate a field that can influence a device in its operation. With this approach, Schmidt and Hutter [34] were able to affect the SRAM content of a microcontroller and successfully injected faults during computations.

3.2 Semi-Invasive Fault Attacks

Compared to non-invasive fault attacks, semi-invasive fault injection strategies require a more sophisticated preparation of the attacked device. Usually, the package of the

chip has to be partly removed in order to provide direct access to the surface of the semiconductor where the entire chip die and its passivation layer remain intact. Once the chip surface is exposed, semi-invasive attacks are performed without additional electrical contact. Classified under the category of semi-invasive fault attacks, Skorobogatov and Anderson [37] introduced the approach of optical fault injection. These kind of attacks are based on the sensitivity of semiconductors to ionization caused by exposing them to intense light sources. The occurring physical effects are based on the formation of electron-hole pairs caused by the absorption of photons within the semiconductor material [38]. In regions of p-n junctions the charge carriers generate a current capable of influencing the state of a transistor. Skorobogatov and Anderson [37] demonstrate that an inexpensive photo flash or an off-the-shelf laser pointer can be sufficient in order to perform optical fault injection. However, modern semiconductor technologies with smaller structure sizes and an increasing number of layers require very accurate optical fault injection techniques. For instance, the wavelength of the light source used, the spot size of a laser beam and the applied energy are important parameters for succeeding in optical fault injection. Trichina and Korkikyan [39] use a yttrium aluminum garnet (YAG) laser for attacking a microcontroller where the optical fault injection setup allowed the precise targeting of the semiconductor circuit in the range of a few μm^2. With the presented approach it is possible to enforce skipped instructions when attacking a specific region with focused laser beams. In general, optical fault attacks can target the front side as well as the rear side of a chip [25]. Removing the package on the front side requires a sophisticated approach including the usage of nitric acid in order to prevent mechanical damage of the chip die. In contrast to that, accessing the chip on its rear side can be achieved in a rather simple way by using a mill for removing the package material. In this case the substrate of the semiconductor is protected from mechanical damage by the heat-sink metal plate which can be removed after milling with pliers. For optical fault injection both strategies have different advantages and disadvantages. As the transistors are placed at the front side, the circuit structure can not be identified when opening the chip on the rear side. Additionally, the substrate has to be penetrated by the laser beam and specific wavelengths are required to reach the sensitive regions. When attacking from the front side, metal layers can hinder the laser beam from hitting a specific target.

3.3 Invasive Fault Attacks

Similar to semi-invasive fault attacks invasive fault injection strategies demand for exposing the chip on its front side. Usually, a silicon oxide or nitride layer covers the semiconductor surface necessary for electrical and chemical protection of the chip. When applying invasive fault attacks, this so-called passivation layer has to be removed in order to gain access to the metal layers of the chip die. Once the interconnect lines of the semiconductor are exposed, the approach of microprobing allows to establish electrical contact to the inner circuit structure of the device. Therefore, specific microprobing equipment is essentially required in order to handle precise positioning of probing needles. Typically, data and address bus lines are a good target choice for microprobing. For instance, extrinsically influencing the value of an address bus allows to access specific memory content of a device. In this context, Skorobogatov [36] provides an intuitive description of the essential steps for performing invasive fault attacks. Beside the fact that fairly expensive and sophisticated equipment is required, invasive fault attacks are the most powerful approaches for fault injection. As the underlying injection mechanism is based on direct

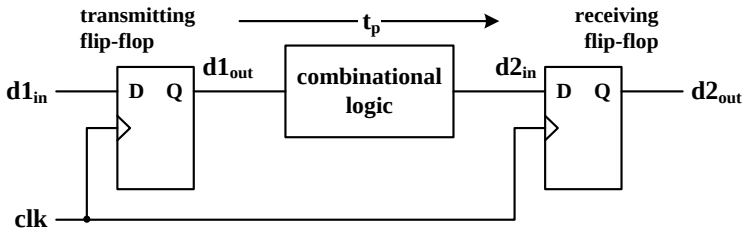

Figure 3.1: Sequential logic of a digital integrated circuit.

electrical contact to the affected target, faults can be induced in a very precise and selective way. However, partial knowledge about the circuit structure is advantageous for proper identification of potential targets on a chip.

3.4 Timing Constraints Violation

In our practical experiments on investigating the influence of non-invasive fault attacks on the behavior of microcontrollers, we apply fault injection using clock glitches and supply voltage manipulation. Both strategies rely on violating the timing constraints for proper operation of the devices in order to provoke erroneous behavior. As illustrated in Figure 3.1, digital integrated-circuits usually consist of two parts. First, combinational logic blocks for data processing and second, storage elements (e.g. D-flip-flops) to synchronize the operations using a common clock signal. Between two rising clock edges, data propagates through the combinational logic causing a propagation delay t_p until the correct output value of the combinational logic can be provided to the input of the receiving D-flip-flop. In addition to the propagation delay t_p, the critical path delay $t_{critical}$ between two flip-flops depends on several other parameters. First, the setup time t_s of the flip-flop defines the interval before the rising clock edge where a stable input value is required. The hold time t_h defines a similar interval but after the rising clock edge is not considered within the critical path delay. Second, the clock-to-output-delay t_{co} describes the interval required to obtain a valid output value by the flip-flop after the rising clock edge. Based on the assumption that routing delays and clock skews between the flip-flops are negligible, the critical path delay can be defined as the sum of t_p, t_s, and t_{co} defined in Equation 3.1. In order to ensure correct operation, the minimal clock period of T_{clk} has to be greater than the maximum critical path delay $t_{critical}$ of the circuit. Additionally, a stable input signal is required during the time intervals t_s before and t_h after the rising clock edge to avoid metastable output behavior of the flip-flop.

$$t_{critical} = t_p + t_s + t_{co} \tag{3.1}$$

By violating the aforementioned timing constraints fault injection is possible either through variations of the supply voltage level or by exceeding the maximal clock period for proper operation. Djellid-Ouar et al. [13] describe the effects of supply voltage glitches on CMOS circuits. Their simulation results show that influencing the combinational logic by applying supply voltage glitches is possible. However, D-flip-flops appeared to be resistant to an attack between two rising clock edges. Additional results on the impact of clock and supply voltage glitches are presented by Zussa et al. [43] and Agoyan et al. [1]. Subsequently, two approaches for influencing the behavior of sequential logic are discussed

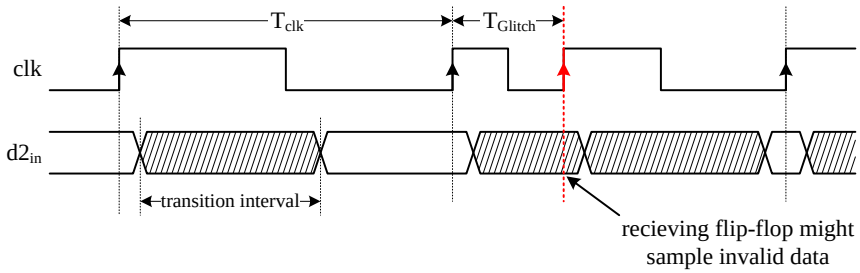

Figure 3.2: Violation of the critical path delay due to clock glitch insertion. The shaded area of $d2_{in}$ represents the unsteady output state of the combinational logic.

by means of the fault injection mechanism for violating the timing constraints using either clock glitch insertion or underpowering.

3.4.1 Clock Glitches

The maximum operating frequency for a synchronous digital device is typically defined by the longest critical path delay. Therefore, the supported clock frequency is defined by a maximum value f_{max} which equals a minimum clock period T_{min} required by the device for correct operation. By decreasing the clock period below T_{min}, the available time between to rising clock edges may not be sufficient for providing the correct output of a combinational logic block. As a result, the register may sample a wrong value if either a stable but incorrect signal level is provided to a register's input at the rising clock edge or input signal transitions during the setup and hold time cause metastable behavior of the register. Both effects can occur if the clock edge arrives before the output of the combinational logic has settled to a stable value. Consequently, tampering with the clock signal allows fault injection by violating the critical path delay $t_{critical}$ of an operation.

In this context, the most intuitive approach for fault injection is overclocking where the clock frequency of a device is increased to a value above the supported operating range. However, faults may be induced during several operations of the device as each cycle is affected by clock frequency modification. Thus, the applied clock frequency represents the only attack parameter for determining the resulting effects. In contrast to overclocking, a precise selection of the affected clock cycle is possible when applying the approach of clock glitch insertion. In this case, the device operates on a nominal clock frequency within the specified operating range. By adding one additional rising edge to the clock signal an additional shortened clock period T_{Glitch} is generated, violating the timing constraints during a specific calculation if $T_{Glitch} < t_{critical}$. Figure 3.2 illustrates the effects of an inserted clock glitch, violating the critical path delay of the combinational logic. The shaded area of $d2_{in}$ represents the unsteady output state of the combinational logic during the transition interval. As it is not guaranteed that stable or correct input data $d2_{in}$ is provided to the register at the rising edge of the clock glitch, an erroneous value may be latched by the register. Based on this strategy, faults can be precisely induced by selecting two attack parameters: the affected clock cycle and the shortened clock glitch period T_{Glitch}.

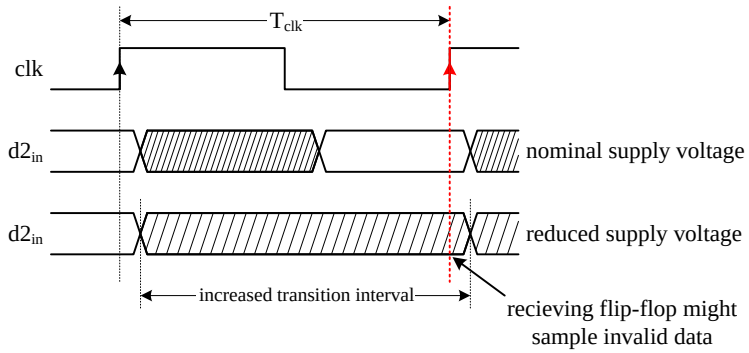

Figure 3.3: Violation of the critical path delay due to underpowering. The shaded area of $d2_{in}$ represents the unsteady output state of the combinational logic.

3.4.2 Underpowering

Provoking timing constraint violations can be additionally achieved by manipulating the supply voltage level. By decreasing the supply voltage level of a device, the propagation delay t_p of the combinational logic block can be increased. This effect relies, among others, on an increased duration for loading the output capacitances of the integrated circuit [43]. In this context, underpowering describes a reduction of the supply voltage to a level below the specified operating range of a device in order to induce faults. Figure 3.3 illustrates the underlying principle of fault induction using underpowering. The propagation delay of a combinational logic is increased to the effect that the critical path delay of an operation is violated and wrong values are sampled by the registers at a nominal clock frequency. Similar to overclocking, permanent underpowering of the device during the whole execution of an algorithm may induce several faults in multiple calculations. In contrast, transient underpowering can be applied by reducing the supply voltage only during specific operations, i.e. during specific time intervals. However, supply voltage changes are limited in time due to unavoidable capacitive effects given, for instance, by parasitic capacitance of supply lines or on-chip decoupling capacitors.

As introduced in Chapter 6, a combination of short-time underpowering and clock glitch insertion is used in our practical experiments in which underpowering only serves the purpose of increasing the propagation delay of an operation without actively inducing faults. As a result, the sensitivity to additionally applied clock glitches can be increased.

Chapter 4

Fault Injection Setup

In order to implement the approach of non-invasive fault injection and to practically perform experiments on microcontrollers, a specific fault injection setup is used. Figure 4.1 depicts a block diagram of the fault injection setup and the connection between the involved components. An FPGA-based fault board acts as the core part of the fault injection setup and provides the clock signal and the power supply voltage for the attacked device, referred to as device under test (DUT). To synchronize an attack between the fault board and the DUT, the trigger and reset signals are used. The attack parameters for the clock glitch injection and the supply voltage manipulation are configured on the control computer. Additionally, the DUT is connected to the control computer for the configuration of a test application and for result communication after an attack. The supply voltage and the clock signal are monitored using an oscilloscope.

In the following chapter, we first introduce the investigated microcontrollers, one Atmel ATxmega 256 and one ARM Cortex-M0. Next, a description of the fault board is given, including its features and attack capabilities. In this context, the clock glitch generation and the supply voltage manipulation are explained. In order to provide a flexible use for a wide range of devices, the fault board provides a specified interface. We designed extension boards for both investigated microcontrollers which can be easily connected to the fault board by utilizing this interface. A description of the two extension boards is given at the end of this chapter.

4.1 Investigated Microcontrollers

The chosen target platforms for our practical experiments targeting non-invasive fault injection are the Atmel ATxmega 256 and the ARM Cortex-M0, for which NXP's LPC1114 implementation of the Cortex-M0 is used. Both microcontrollers are largely used in a wide range of of applications. Although they are not dedicated to secure devices, they might be used in security applications due to a huge number of supported features, including, for example, the hardware support for AES encryption on the ATxmega 256. Moreover, analyzing the vulnerability to fault injection of two different microcontroller platforms allows a precise characterization and comparison of the occurring effects. Thus, it is possible to identify common weak points which should be considered in general for security-relevant applications. Following, we briefly introduce the ATxmega 256 and the Cortex-M0 by their features and properties, with a focus on architectural details, particularly relevant in our experiments.

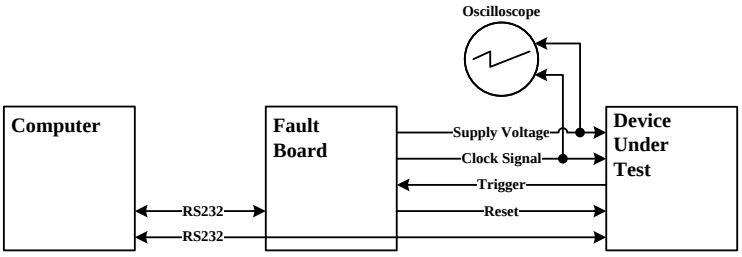

Figure 4.1: Block diagram of the fault injection setup.

4.1.1 Atmel ATxmega 256

The Atmel ATxmega 256 is a low-power 8/16-bit microcontroller based on a Harvard architecture, meaning that separated memories and buses are used for program memory and data memory. The program is stored in an internal flash memory, which is accessible via the program interface of the ATxmega 256 or by the software application executed on the device. The data memory space linearly maps the I/O space starting at address 0x0, followed by the internal Electrically Erasable Programmable Read-Only Memory (EEPROM) and the internal Static Random-Access Memory (SRAM). The reduced instruction set computer (RISC) architecture of the ATxmega 256 supports 142 instructions, complying with the Atmel AVR instruction set. Most of them execute in a single clock cycle. Additionally, the 16-bit program memory bus allows to load a 16-bit instruction in a single clock cycle resulting in a two-stage pipeline with one *fetch stage* and one *execution stage* as shown in Figure 4.2. Based on this pipeline structure, one 16-bit instruction is loaded from the program memory while the previously loaded instruction is executed simultaneously. Thirty-two 8-bit general purpose registers are available on the ATxmega 256where the arithmetic logic unit (ALU) can access two registers during the execution of an instruction in a single clock cycle. For addressing of program memory or the data memory space, 6 general purpose registers can be used. In this case, two registers are combined to provide a 16-bit memory address.

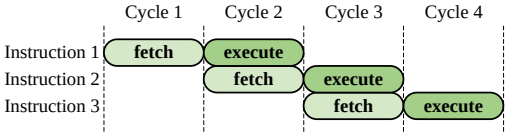

Figure 4.2: Pipeline diagram of the Atmel ATxmega 256.

For our practical experiments we use the ATxmega 256A3 in a 64-pin thin quad flat package (TQFP). This version of the ATxmega 256 series provides 256 KiB flash memory, 16 KiB SRAM and 4 KiB EEPROM. The maximal operating frequency is 32 MHz which equals a minimal clock cycle duration of 31.25 ns. The required supply voltage is specified between 1.6 V and 3.6 V with at least 2.7 V recommended when using a clock frequency above 12 MHz. The available option of using an external clock source actually makes it possible to induce clock glitches. Particularly interesting features of the ATxmega 256A3 are the hardware support for AES and a hardware multiplier. The so-called AES crypto engine supports a key size of 128 bits and requires 375 clock cycles for the encryption or decryption of a 128-bit data block. The hardware support for multiplication has a capability of multiplying two 8-bit register entries. After two cycles for execution, the

16-bit result is written to two dedicated 8-bit general purpose registers. Both features are considered in our clock glitch attacks. For further information about the ATxmega 256 we refer to the datasheet [6] and the user manual [5].

4.1.2 ARM Cortex-M0

The ARM Cortex-M0 represents the smallest available ARM processor based on the ARMv6-M architecture. The low-power 32-bit processor implements a Von-Neumann architecture, using the same 32-bit bus system for accessing the program and the data memory. The program code is stored in an internal flash memory starting at address 0x0 in the linearly mapped memory address space followed by the internal SRAM. The Cortex-M0 implements a RISC architecture and supports 56 instructions, most of them correspond to the ARM *Thumb* instruction set. Only a few instructions supported by the Cortex-M0 belong to the *Thumb-2* instruction set which includes additional 32-bit instructions. However, most operations of the Cortex-M0 are performed with 16-bit instructions. The instruction execution procedure of the processor is based on a three-stage pipeline with one *fetch stage*, one *decode stage*, and one *execution stage*. Figure 4.3 depicts the pipeline structure for executing 16-bit instructions. The *fetch stage* represents a conspicuous part of the pipeline. One fetch operation is performed in every second clock cycle, where two 16-bit instructions are simultaneously fetched from program memory. As a consequence, one instruction remains for either three or four clock cycles in pipeline although instruction decode and execute is performed in every clock cycle. For data operations, the Cortex-M0 provides thirteen 32-bit general purpose registers. Most of the instructions only operate on the so-called low register (*R0* to *R7*). For memory addressing, the 32-bit address can be specified by the value of one general purpose register. Furthermore, several address modes are available to support application-specific requirements.

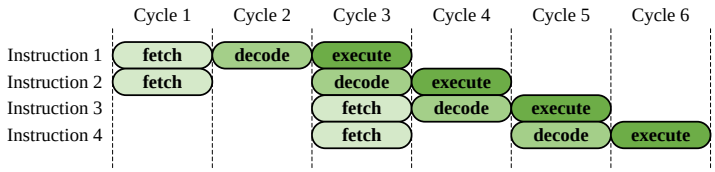

Figure 4.3: Pipeline diagram of the ARM Cortex-M0.

For our practical experiments we use NXP's implementation of the ARM Cortex-M0, namely LPC1114FN28, available in a 28-pin dual inline package (DIP). The internal memories are a 32 KiB flash memory for program code and a 4KiB SRAM. The clock generation unit of the LPC1114 allows to bypass all internal clock sources in order to use an external clock signal which is necessarily important for clock glitch injection. The maximal operating frequency is 50 MHz which equals 20 ns for the minimal clock cycle duration. Regardless of the used clock frequency, the supply voltage level can be chosen between 1.8 V and 3.6 V. In contrast to the ATxmega 256, the LPC1114 does not support hardware acceleration for cryptographic primitives. The integrated hardware multiplier is capable of multiplying two 32-bit values into a 32-bit result in a single clock cycle. The most significant 32-bits of the result are discarded. For further information about the ARM Cortex-M0 we refer to the user guide [2]. Details about NXP's LPC1114 can be found in the data sheet [30] and the user manual [31]. For the rest of this work the name Cortex-M0 implies the LPC1114FN28 implementation from NXP.

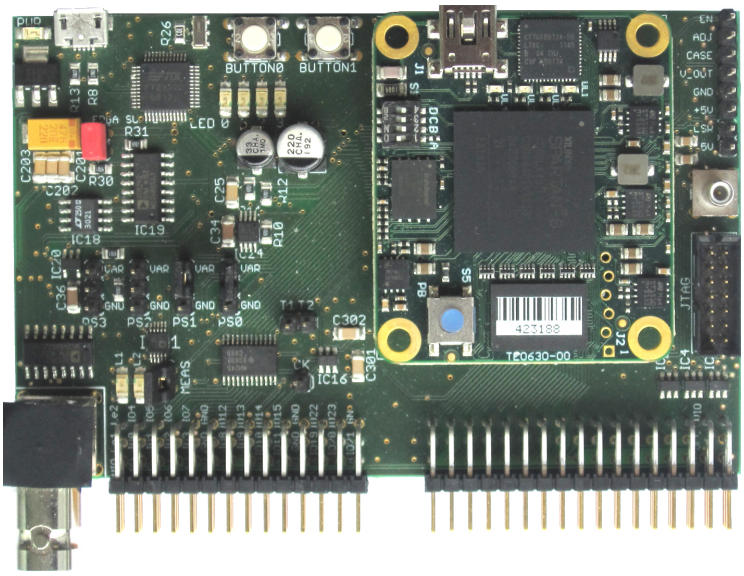

Figure 4.4: Fault board of the fault injection setup.

4.2 Fault Board

The main part of our fault injection setup is a custom-made device for tampering with the supply voltage and the clock signal. This so-called fault board is depicted in Figure 4.4. A *XILINX Spartan-6 XC6SLX45* FPGA is the main part of the fault board, providing a flexible approach to control the entire attack procedure. Additionally, the FPGA is responsible for clock glitch generation and the flow control of supply voltage manipulation during an attack. The fault board provides a specified interface for connecting an investigated target. Therefore, extension boards are used which have to be individually designed for each device under test (DUT). In order to address a wide range of devices, the fault board provides a huge number of configurable I/O pins in addition to several dedicated pins for performing an attack. These dedicated pins are: the clock and power supply, the trigger input, the serial communication interface, and the reset output. As illustrated in Figure 4.1, the following signals are used for our experiments, targeting the ATxmega 256 and the Cortex-M0:

- **Supply voltage.** The supply voltage for the extension board is provided by the fault board. Thus, configurable supply voltage manipulation is possible.
- **Clock signal.** The clock signal including the configurable clock glitch insertion for an attack is generated by the FPGA of the fault board and can be directly applied to the DUT.
- **Trigger signal.** The trigger input of the fault board allows to synchronize an attack between the fault board and the test application running on the microcontroller.
- **Reset signal.** The reset signal is used to put the microcontroller into an initial state at the beginning of an attack.

A control computer is used for configuration of the fault board and for communication with the DUT. The fault board provides an universal serial bus (USB) interface with an

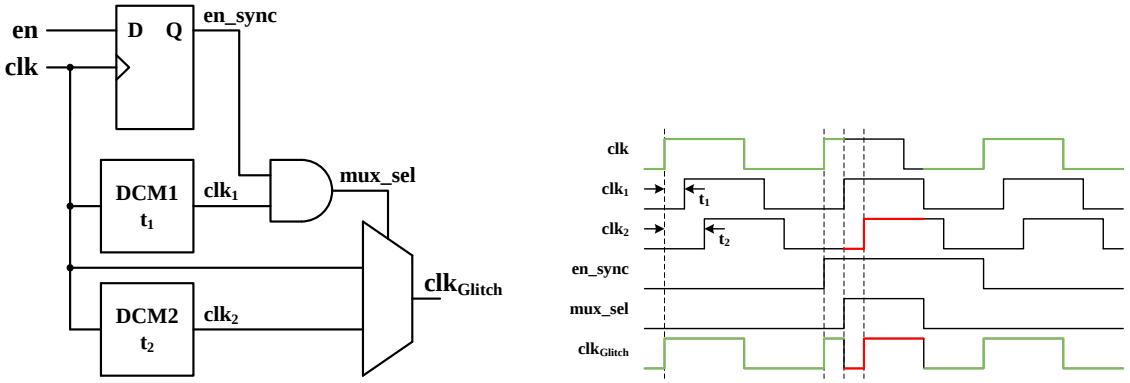

Figure 4.5: Block diagram and timing chart of the clock glitch generation unit.

USB to serial converter, providing two separate serial interfaces. One of the serial interfaces is connected to the FPGA. The second serial interface is used to communicate with the DUT and therefore connected to the interface for the extension boards. Furthermore, the power supply for the fault board is provided by the USB connector. Configuration and communication via the serial interfaces are done via MATLAB scripts on the control computer.

4.2.1 Clock Glitch Generation

The clock glitch generation unit of the fault board is based on the approach presented by Endo et al. [14]. In doing so, two phase-shifted clock signals clk_1 and clk_2 are derived from the nominal clock signal clk resulting in three signals with different phases. As shown in Figure 4.5, t_1 and t_2 define the delays between the nominal clock signal and the phase-shifted clock signals clk_1 and clk_2 on condition of $t_1 < t_2$. By switching between these signals it is possible to generate a clock signal clk_{Glitch} with an additional rising edge. The functional principle of the clock glitch generation unit is depicted in Figure 4.5, including a block diagram and the corresponding timing chart for the clock and control signals. Two digital clock manager (DCM) of the *XILINX Spartan-6* FPGA are used to shift the phase of the nominal clock signal and to generate clk_1 and clk_2. The DCM units are part of the FPGA clocking resources and provide a variable phase shift mode for adjusting the phase shift of a clock signal during operation using a step size between 10 ps and 40 ps. Further information about the DCM units can be found in the user guide for clocking resources of the *XILINX Spartan-6* FPGA in [40]. A multiplexer is used to select either the nominal clock signal clk or the shifted clock signal clk_2 generated by DCM2. The output signal of the multiplexer represents the glitched clock signal clk_{Glitch}. DCM1 is used to generate the shifted clock signal clk_1 which is further combined with the synchronized enable signal en_sync. The resulting select signal mux_{sel} is used to control the multiplexer. The enable input signal en is synchronized with the nominal clock signal clk using a D-flip-flop. Without inserting a clock glitch, the en input is low and the multiplexer forwards the nominal clock signal clk to its output. In case of inserting a clock glitch, the first rising edge of the affected clock cycle is still defined by the nominal clock signal clk. As soon as the enable signal en_sync is set to high-level with the rising edge of clk, the multiplexer switches to clk_2 at the rising edge of clk_1. The resulting clock signal clk_{Glitch} is set to low-level due to the fact that clk_2 is still low at this point. The second rising edge of clk_{Glitch} is defined by the rising edge of clk_2. At the falling edge of

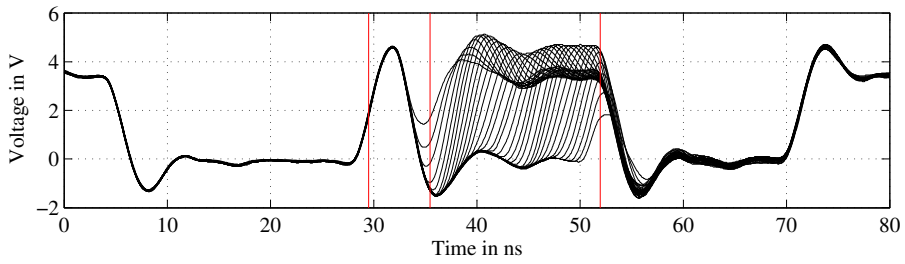

Figure 4.6: Waveforms of the clock signal with a step width of 0.5 ns for T_{Glitch} between 5 ns and 18 ns.

clk_1 the select signal mux_{sel} is set to low-level and the multiplexer forwards the nominal clock signal clk again.

The fault board offers several configurable options for clock glitch generation. The nominal clock frequency for the DUT can be chosen between 4 MHz and 120 MHz. A trigger signal is used to synchronize clock glitch insertion and to precisely select the attacked state of the DUT. The fault board allows to configure the number of clock cycles between the trigger event and the glitch insertion as well as the number of consecutive clock cycles manipulated with a glitch. Furthermore, the phase shifts for clk_1 and clk_2 can be configured to specify the position of the additionally inserted falling and rising edge within a nominal clock cycle.

For the clock glitch generation in our practical experiments we use a constant value for clk_1 when clk_2 is varied according to the desired value for T_{Glitch}. The clock glitch period T_{Glitch} is defined by the delay between the nominal rising edge of the clock signal and the additionally inserted rising edge of the glitch. A nominal clock frequency of 24 MHz ($T \approx 41.7$ ns) is used to clock both investigated microcontrollers, the Cortex-M0, and the ATxmega 256. For this clock frequency, T_{Glitch} can be chosen approximately between 5 ns and 18 ns. The minimal step width for T_{Glitch} of approximately 40 ps is defined by the DCM units of the *XILINX Spartan-6* FPGA. It is notable that the value of T_{Glitch} between 5 ns and 18 ns corresponds to an equivalent frequency between 55 MHz and 200 MHz which is above the supported clock frequency range of both microcontrollers. Figure 4.6 depicts the waveforms of the clock signal with a glitch period between 5 ns and 18 ns and a step width of approximately 0.5 ns. T_{Glitch} is measured between the positive edges of the clock signal at a voltage level of 1.65 V.

4.2.2 Supply Voltage Manipulation

The supply voltage generation circuit of the fault board allows to select different supply voltage levels for the DUT via a 4-to-1 multiplexer. Four linear voltage regulator are available on the fault board to provide configurable voltages between 0 V and 5 V. In order to configure the voltage levels, digitally controlled variable resistors are used. To tamper with the supply voltage of the attacked device, the multiplexer allows to switch between the four pre-configured voltage sources. The resistors and the multiplexer are both controlled by the FPGA. As for the clock glitch insertion, the trigger signal is used to synchronize the manipulation of the supply voltage with the DUT. A configurable delay between the trigger event and the point in time of changing the supply voltage allows a precise selection of the attacked state of the DUT. Thus, it is possible to change the supply voltage of the attacked device during operation as the fault board additionally supports a

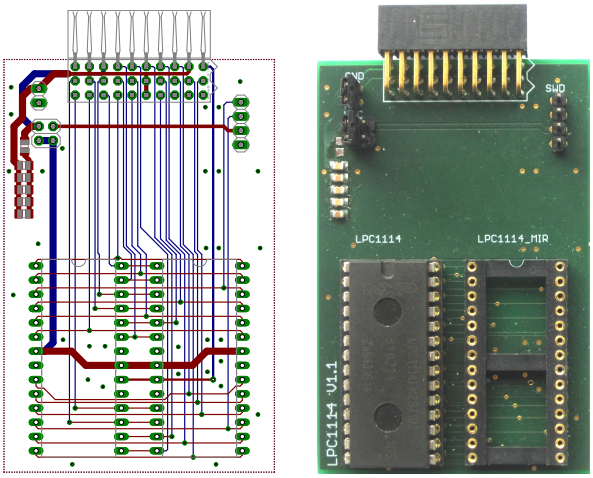

Figure 4.7: Extension board for the Cortex-M0.

configurable duration for the voltage manipulation.

In our practical experiments, a nominal supply voltage of 3.3 V is used for the Cortex-M0 and the ATxmega 256. The first of the four configurable voltage sources is set to 3.3 V and selected via the multiplexer to power the DUT. For underpowering, the second voltage source is set to the underpowering voltage U_{Glitch}. As the same trigger event is used for clock glitch insertion and supply voltage manipulation, the voltage reduction is synchronized with the inserted clock glitch during an attack by using the configurable delay and duration. Further details about our approach of combining clock glitch attacks with additional underpowering are given in Chapter 6.

4.3 Extension Boards

Investigated targets are connected to the fault board using custom-made extension boards. The specified interface of the fault board provides a variety of signals supporting fault injection, synchronization and communication with the device under test. Figure 4.7 and Figure 4.8 depict the extension boards and the corresponding PCB layouts for the Cortex-M0 and the ATxmega 256, respectively. Several signals are equally used on both extension boards. The receive and transmit lines of the serial interface are directly connected to the corresponding pins of the microcontroller using the internal Universal Asynchronous Receiver and Transmitter (UART) module. An output pin of the microcontroller is connected to the trigger input of the fault board to allow software-controlled trigger events for synchronized fault injection. Furthermore, the reset signal of the MCU is controlled by the fault board. The configurable I/O pins of the fault board are connected to free I/O ports of the microcontroller even though they remain unused in our fault injection setup. The clock signal of the clock glitch generation unit is directly connected to the external clock input pin of the microcontroller. The built-in clock prescaler and phase-locked loop (PLL) are deactivated. On both extension boards, the decoupling capacitors are separately placed in parallel to additionally support precise power consumption measurement and to minimize distortions due to capacitive smoothing. This circuit arrangement is particularly important for differential power analysis (DPA) in which the varying power consumption of a device is analyzed to retrieve information about the performed oper-

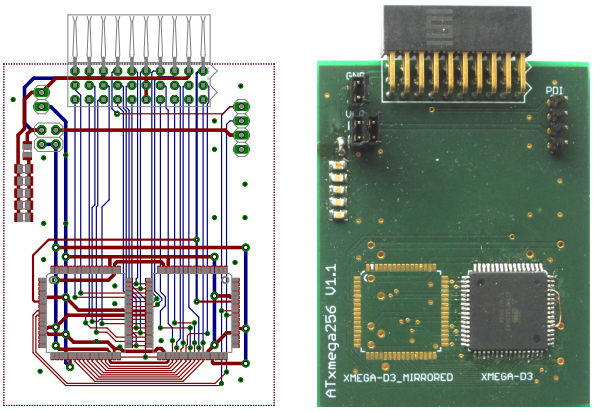

Figure 4.8: Extension board for the ATxmega 256.

ations. Furthermore, disconnecting the capacitors from the supply voltage line during underpowering attacks is necessary in order to provide accurate timing and acceptable rise and fall times when changing the supply voltage level. As illustrated in Figure 4.7 and Figure 4.8, the extension boards allow a bottom-up placement of the microcontrollers which is particularly important for invasive or semi-invasive attacks where access to the bottom side of a device is advantageous for performing an attack (e.g. rear-side laser fault injection). On both extension boards, the interfaces for programming the microcontrollers are accessible via a pin header. For programming and debugging of the Cortex-M0, the 2-pin serial wire debug (SWD) interface is used. In case of the ATxmega 256, we use Atmel's 2-pin program and debug interface (PDI). Additionally, it is possible to power the extension boards with the supply voltage provided by the programmer in order to allow programming and debugging without a connection to the fault board.

Chapter 5

Instruction Set: Attacks

The possible vulnerability of microcontrollers to non-invasive fault attacks depends on a variety of reasons including the central processing unit (CPU) structure and the corresponding instruction execution procedure as well as the applied attack parameters. In order to allow detailed insight into the resulting erroneous behavior of the ATxmega 256 and the Cortex-M0, the effects of clock glitch attacks are investigated on different instructions for both MCUs. In this chapter the investigated instructions are introduced, including a detailed description of the applied attack scenario for our practical experiments. Additionally, possible effects of injected faults on the instruction execution procedure of microcontrollers are discussed.

5.1 Expected Fault Behavior

When applying clock glitch attacks on the execution procedure of instructions, two important characteristics have to be considered for analyzing the resulting behavior of an MCU. First, the execution process of an instruction is separated into several stages based on the pipeline architecture of the CPU. Second, results vary depending on the glitch period T_{Glitch} which defines the delay between the actual and the additionally inserted rising clock edge. Thus, a specific state of the combinational logic is hit by the attack. Based on the pipeline structure of the MCU, the execution process of an instruction is commonly separated into the *fetch stage*, the *decode stage* and the *execution stage*. During the fetch stage, the instruction is loaded from program memory into an instruction fetch buffer. A fault induced during this stage could possibly result in different or wrong instructions, either by reading from a wrong memory address or by reading faulty data. Similar effects occur if the instruction is correctly fetched but misinterpreted in the decode stage. In this stage, a specific operation of the MCU is selected according to the opcode of the instruction. Additionally, the involved operands are selected as specified within the instruction. Again, it might be possible that different or wrong instructions are decoded or wrong operands are selected. Finally, the execution stage is responsible for performing the operation on selected operands or memory addresses. A fault in this stage might lead to wrong calculations and wrongly or not updated registers or memory values. In case of conditional branches, the program flow might be modified. In order to compare the resulting behavior after an attack of the ATxmega 256 and the Cortex-M0 and to draw meaningful conclusions, we focus on three classes of instructions for both MCUs: *arithmetical/logical instructions*, *branch instructions* and *memory instructions*. According to

Table 5.1: Investigated instructions of the ATxmega 256 and the Cortex-M0.

Instruction Class	ATxmega 256	Cortex-M0
Arithmetical/Logical	`add Rd,Rn`	`adds Rd,Rn`
	`mul Rd,Rn`	`muls Rd,Rn`
		`lsls Rd,#imm`
Memory	`ld Rd,X`	`ldr Rd,[Rn]`
	`st X,Rn`	`str Rd,[Rn]`
Branch	`breq label`	`beq label`

these three categories we have chosen a set of instructions, as summarized in Table 5.1 and introduced in the following section.

5.2 Investigated Instructions

Both MCUs, the ATxmega 256 and the Cortex-M0, implement a RISC architecture in which the instruction set is commonly separated into instructions for data handling, instructions for data processing and instructions for program flow modification. Data handling instructions include operations for transferring data from or to registers and memory locations. The group of data processing instruction provide arithmetical operations usually performed on register data: addition, subtraction, multiplication, and division. In addition, logical instructions for bitwise operations on data are assigned to this group. For program flow modification, branch or conditional branch operations allow to change the location of the program code execution. Based on this categorization, we selected comparable instructions for both MCUs and again defined three groups: *arithmetical/logical instructions*, *branch instructions* and *memory instructions*. Table 5.1 gives an overview of the attacked and investigated instructions. Regardless of the MCU architecture, each instruction is described by the opcode and additional bits for the operands. The opcode relates to a specific operation of the MCU where the involved operands can be a register, a constant value or additional parameters for the instruction.

5.2.1 ATxmega 256

All investigated instructions of the ATxmega 256 are based on the Atmel AVR 16-bit instruction set architecture. As stated in the instruction set manual for the ATxmega 256 in [4], the selected instructions can be described by the following features and attributes.

- `add Rd,Rn`. An arithmetic addition of two 8-bit values in registers `Rd` and `Rn` is performed by this instruction. After the operation, the result equals the value of the destination register `Rd`. The two registers `Rd` and `Rn` are specified within the instruction, each by 5 bits. Therefore, `Rd` and `Rn` can be individually chosen from the 32 available general purpose registers. The `add` instruction requires one clock cycle for execution.

- `mul Rd,Rn`. This instruction performs a multiplication of the two 8-bit values in registers `Rd` and `Rn`. The 16-bit product is stored in register `R0` (low byte) and in

register `R1` (high byte). The two registers `Rd` and `Rn` can be chosen individually from the 32 available general purpose registers. The `mul` instruction requires two clock cycles for execution.

- `breq` *`label`*`.` A conditional branch is performed by testing the zero flag of the status register of the MCU. The instruction `breq` should be immediately executed after an instruction which influences the zero flag. In our test application, `cp Rd,Rn` is used to perform a compare between two registers. If the values of `Rd` and `Rn` are equal, the zero flag is set and the consecutively executed `breq` instruction branches relatively to the program counter corresponding to the given value of `label`. The operand `label` is given as operand within the instruction and represents the offset between -63 and +64 from the actual value of the program counter. The `breq` instruction requires two clock cycles if the condition is true and the branch is executed. Otherwise, one clock cycle is required.

- `ld Rd,X.` This instruction loads one byte indirectly from the data space, e.g. the internal SRAM of the MCU, to the register `Rd` using a 16-bit address pointer `X`. Two 8-bit general purpose registers specify the value of `X` whereas `R26` defines the low byte and `R27` the high byte of the address. The destination register `Rd` can be chosen individually from the 32 available general purpose registers including the two address registers. The `ld` instruction requires two clock cycles for execution when accessing the internal SRAM.

- `st X,Rn.` With `st X,Rn` one byte of data given by the register `Rd` is stored to the data space of the MCU. As for the equivalent load instruction, indirect addressing is used. In this case, `X` defines the 16-bit destination memory address. Again, each of the 32 general purpose registers can serve as source register `Rd`. In our test application, data is written to the internal SRAM using this instruction in which one clock cycle is required for execution.

5.2.2 Cortex-M0

The instructions selected for the Cortex-M0 are based on the ARM Thumb 16-bit instruction set architecture. The user guide of the Cortex-M0 in [2] describes the available instructions of the Cortex-M0. In the following, the investigated instructions are introduced.

- `adds Rd,Rn.` This instruction performs an arithmetic addition of the values given by two 32-bit general purpose registers `Rd` and `Rn`. The result is written to the destination register `Rd`. The applicable registers are restricted to `R0` to `R7`. The `adds` instruction requires one cycle for execution.

- `muls Rd,Rn.` The `muls` instruction multiplies the values specified by the two 32-bit general purpose registers `Rd` and `Rn`. The least significant 32 bits of the product are written to the destination register `Rd`. The instruction is again restricted to `R0-R7`. ARM provides two possible hardware implementations for the `muls` instruction, either requiring 32 cycles or one cycle for execution. In our practical experiments we use NXP's LPC1114 implementation which provides the one-cycle multiplier for this instruction.

- `lsls Rd,#imm.` In contrast to the ATxmega 256, we additionally analyzed the `lsls` instruction of the Cortex-M0 which performs a logical left shift operation on the 32-bit register `Rd`. The shift length between 0 and 31 bits is defined by the constant value *`#imm`*. After the operation, the result equals the value of `Rd`. For the execution of `lsls` one cycle is required.

- `beq` *`label.`* The `brq` instruction performs a conditional branch to program counter relative address specified by the operand *`label`*. The condition for executing the branch depends on the value of the zero flag in the application program status register of the MCU. A previous executed instruction is used in order to define the condition. In our test application, `cmp Rd,Rn` is used to compare the values of two registers. The offset to the program counter given by *`label`* can vary between -256 and +255 in case of conditional branching. The `beq` instruction requires either three clock cycles if the condition is true and the branch is executed or one clock cycle if the branch is not taken.

- `ldr Rd,[Rn].` This instruction loads the register `Rd` with the 32-bit value from the given memory address by `Rn`. The used registers are restricted to `R0-R7`. The `ldr` instruction requires two cycles for execution.

- `str Rd,[Rn].` The `str` instruction stores a 32-bit value at the memory address defined by the register `Rn`. As for the similar load instruction, the registers are restricted to `R0-R7` and the execution requires two clock cycles.

5.3 Attack Scenario

This section describes the attack procedure used for analyzing the influence of fault attacks on the instruction execution procedure of the ATxmega 256 and the Cortex-M0. The fault injection setup is based on the fault board and the two extensions boards for the investigated MCUs presented in Section 4. Regardless of the attacked MCU platform, a test application is written in mixed C and assembly language for each instruction. Inline assembly is used for the definition of the attacked instruction and for the initialization of general purpose registers. To guarantee an unmodified instruction execution sequence of the inline assembly block, automated code optimization is disabled for the attacked code parts. Additionally, the test application is used to perform communication with the control computer and to synchronize the attack with the fault board. The control computer communicates with the microcontroller and the fault board using UART interfaces. As illustrated in Figure 5.1, the test application and the operations performed by the control computer comply with the following attack procedure:

- **Initialization.** In the initialization phase, the control computer configures the fault board to define the desired attack parameters as given in Section 4.2, including the glitch period T_{Glitch} as well as the number of clock cycles between the trigger signal is set and the clock glitch is injected. Additionally, the control computer requests the fault board to reset the MCU. After the reset, the MCU starts an initialization routine to configure the system clock for using the clock signal provided by the fault board. The I/O port configuration is set for the trigger output signal and the UART interface is initialized for communication with the control computer. After

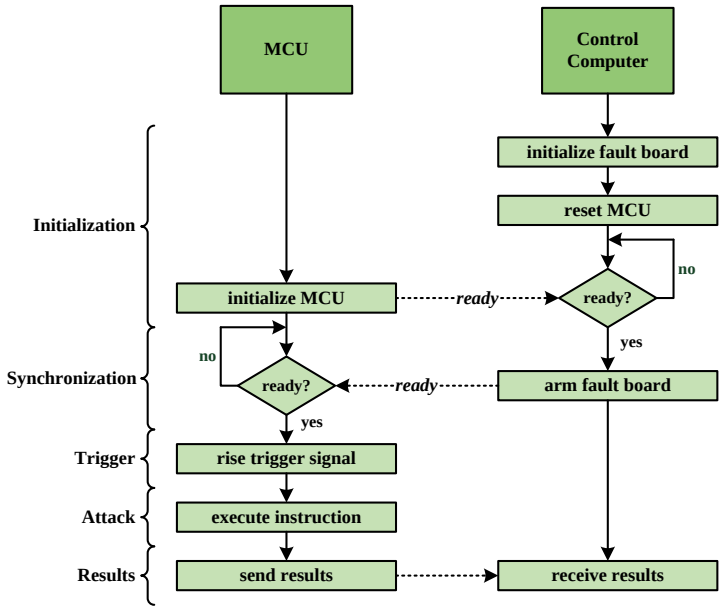

Figure 5.1: Sequence of operations performed during the clock glitch attack of an instruction on the ATxmega 256 or the Cortex-M0.

this procedure, the MCU signals this state to the control computer using the UART interface.

- **Synchronization.** The synchronization stage starts after the test application has finished the initialization routine of the MCU. The control computer prepares the fault board for an attack. In this context, the clock glitch unit is armed, meaning the fault board is sensitive to its trigger input. The execution of the test application is resumed as soon as the fault board is ready and signals this state to the MCU using again the UART interface.

- **Trigger signal.** To ensure a cycle accurate synchronization between the fault board injecting the clock glitch and the microcontroller performing the instruction which should be hit by the attack, the microcontroller rises its trigger output signal. According to the attack settings of the fault board, the clock glitch is inserted after the predefined number of clock cycles after the trigger event.

- **Instruction execution.** Immediately after the trigger signal is set by the microcontroller, the test application starts with the execution of the inline assembly code block. The affected instruction is then executed in compliance with the time interval between the trigger event and clock glitch injection. To keep this time interval constant and independent from other necessary assembly instructions, which are executed between the trigger signal being set and the attacked instruction being executed, `nop` instructions are used. Additionally, the attacked instruction is surrounded by `nop` instructions to ensure proper execution of all other code parts and to avoid any unpredictable side effects.

- **Result communication.** In the final step, all accessible CPU and general purpose register values are transferred to the control computer and are further used for result verification.

For each investigated instruction, a reference execution is performed using the same execution procedure but without inserting a clock glitch. Based on the approach of differential fault analysis (DFA), the results of the reference execution are compared with faulty results after an attack was performed. To analyze the effects of clock glitches on different execution stages of an instruction (fetch, decode, execute), attacks are repeated for each instruction with different time intervals between the trigger signal is set and the clock glitch is inserted. The clock period is only tampered within one clock cycle to prevent simultaneously influencing several pipeline stages by one attack. Additionally, the impact of the glitch period T_{Glitch} on the resulting behavior of the MCU is observed by repeating the attack for each instruction and again for each pipeline stage with different values for T_{Glitch}. Based on the different erroneous values and register entries after an attack, it is possible to retrieve information about the affected hardware parts of the MCU and how clock glitches can be used to manipulate the instruction execution procedure.

Chapter 6

Instruction Set: Results

This chapter describes in detail the obtained erroneous behavior of the Cortex-M0 and ATxmega 256 in consequence of injected faults. For both microcontrollers, we present common and individual results for the attacked instructions listed in Table 5.1. The influences of clock glitches on the attacked execution stage of an instruction are analyzed individually. Additionally, we discuss the impact of the clock glitch period T_{Glitch} on a corresponding erroneous result. To characterize the effects of clock glitch attacks for the Cortex-M0 and the ATxmega 256, the practically performed fault injection experiments are based on a black-box scenario as no detailed information about the hardware implementation of the MCUs is available. In this context, we particularly aim at determining attack procedures and parameters for retrieving reproducible and understandable results. The presented results should serve as a basis for characterizing the MCU hardware implementation and further for defining fault models in order to point out weak spots of the MCUs. In case of using the Cortex-M0 and the ATxmega 256 for security relevant applications, the identified vulnerabilities should be necessarily considered in particular. In the following, we describe the obtained results for the Cortex-M0 and the ATxmega 256, including the applied attack parameters and the attacked test applications for each instruction listed in Table 5.1. Furthermore, the results for both MCUs are compared to identify common relations in their erroneous behavior caused by the applied clock glitch attacks.

6.1 Cortex-M0

All attacks performed on the instruction set of the Cortex-M0 are based on the common attack scenario presented in Section 5.3. After each attack, the values of the 32-bit general purpose registers $R0$ to $R12$ are transferred to the control computer. Additionally, the values of the stack pointer ($R13$), the link register ($R14$), and the program counter ($R15$) are observed. As shown in Figure 4.3, the pipeline of the Cortex-M0 consists of three stages, namely *fetch stage*, *decode stage* and *execution stage*. If only 16-bit instructions are used, which is the case for all instructions investigated in our practical experiments, two instructions are loaded from program memory into the instruction fetch buffer with a single fetch operation. Considering the following decode and execution stage, the first instruction is consecutively decoded and executed after the fetch stage. In contrast, the second instruction is decoded and executed one clock cycle later. The delay cycle between the fetch and decode stage for the second instruction causes a diversity in the number of

clock cycles while an instruction stays in the pipeline. This effect has to be considered when trying to insert a clock glitch during a specific pipeline stage of an instruction. In our test applications, `nop` instructions are used for positioning the attacked instruction within the pipeline, i.e. selecting whether the investigated instruction is the first or the second one of the two 16-bit instructions fetched in one cycle. To ensure a constant time interval between the trigger event released by the microcontroller and the clock glitch inserted by the fault board, the `nop` instructions are placed in the source code before the trigger output pin is set.

6.1.1 Underpowering

First attacks targeting the Cortex-M0 showed 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 an inconsiderable 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 6.1, the supply voltage of the Cortex-M0 is reduced from 3.3 V in normal operation to $U_{Glitch} = 1.2\,\mathrm{V}$. This value is beyond the minimum operating voltage of 1.8 V given in the datasheet [30]. However, further experiments show that by only applying underpowering without inserting a clock glitch, the operation of the MCU is not influenced. Only after additionally inserting a clock glitch while the supply voltage level is reduced, fault injection is possible. In this way, it is ensured that the value of the clock glitch period T_{Glitch} is primarily responsible for the resulting erroneous behavior of the MCU.

In order to enable short-time underpowering on the Cortex-M0 extension board, the decoupling capacitors between supply voltage and ground are not assembled. However, changing the supply voltage requires several clock cycles until the actual voltage level reaches the desired steady state, as shown in Figure 6.1. The reason for this effect are parasitic capacities of the PCB and the integrated circuit of the MCU which have to be recharged in case of changing the supply voltage level. Therefore, it is necessary to tamper with the supply voltage several clock cycles before the clock glitch is inserted. Additionally, accurate timing for attacks, inducing faults only by tampering with the supply voltage would be rather difficult.

In contrast to only inserting clock glitches, additional underpowering might be detected by an optional applicable brown-out detection (BOD) mechanisms provided by the used Cortex-M0 implementation, NXP's LPC1114. The brown-out detection monitors the supply voltage level by comparing it to fixed reference voltage. The response characteristic of the BOD is specified by the brown-out detection time t_{BOD} which defines the minimal duration, while the voltage has to be below the predefined voltage level to detect a supply voltage deviation. This feature usually allows microcontrollers to react to unreliable power supply or flat batteries in order to bring the program or the application into a save state. However, the brown-out detection might also be used as a countermeasure against underpowering or supply voltage glitch attacks. To detect short-time underpowering attacks with the brown-out detection, t_{BOD} has to be below the duration of underpowering necessary for the attack. The LPC1114 provides two strategies for handling detected deviations of the supply voltage. First, the brown-out interrupt to call a user-defined service routine and second, the brown-out reset to restart the MCU. In case of an BOD reset, the corresponding flag of the system reset status register is set. Thus, the reset source is

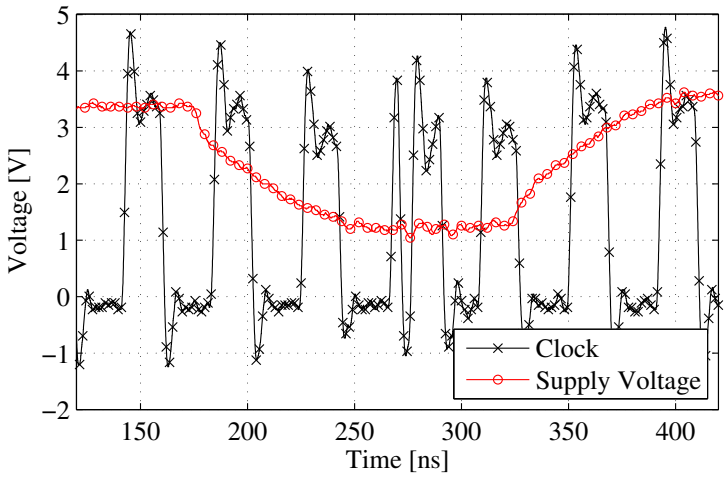

Figure 6.1: Clock signal and supply voltage of the Cortex-M0 during an attack ($T_{Glitch} = 10.0\,\text{ns}$, $U_{Glitch} = 1.2\,\text{V}$).

defined and can be determined by an application after the MCU is initialized again.

In our tests, we analyzed the response characteristic of the brown-out detection when using either the brown-out interrupt or the brown-out reset. For both options, different threshold voltages can be selected. The most sensitive configuration of the brown-out detection is given for the highest selectable threshold voltage. This implies that the threshold voltage is set to $U_{th,int} = 2.8\,\text{V}$ in case of analyzing the brown-out interrupt and to $U_{th,rst} = 2.71\,\text{V}$ for the brown-out reset. To get independent results for both BOD options, the brown-out interrupt and the brown-out reset are individually tested. In each test run, short-time underpowering is applied to the Cortex-M0 by modifying the supply voltage according to the voltage characteristics and underpowering duration as shown in Figure 6.1. Therefore, only the supply voltage is changed without inserting clock glitches. Moreover, the supply voltage is reduced by 0.1 V starting at the nominal supply voltage of 3.3 V. In case of detecting the applied supply voltage reduction by the BOD mechanism, the corresponding BOD event is signaled to the control computer. The state of the BOD reset flag and the execution of the interrupt service routine, respectively, allow to distinguish between BOD reset and BOD interrupt.

Results show that the brown-out interrupt service routine is correctly executed in a range of $1.0\,V \leq U_{Glitch} < U_{th,int}$. Instead of executing the interrupt service routine, two effects are observed if $U_{Glitch} < 1.0\,\text{V}$. In about 10 % of repeatedly performed test cases with $U_{Glitch} < 1.0\,\text{V}$, the actual program flow continued. In the remaining 90 %, a hard fault exception occurred. In this case, the normal program flow is interrupted and the hard fault handler is executed. Similar to an interrupt service routine, the hard fault handler can contain a user-defined code for handling the exception. Compared to all other exceptions and interrupts, the hard fault exception is treated with the highest priority level. Reasons for hard exceptions are either memory-related faults (e.g. bus errors) or program usage faults (e.g. execution of an invalid instructions) as stated in [42]. For additionally analyzing the response characteristic of the brown-out detection in respect of the detection time t_{BOD}, further tests with different underpowering durations are performed. The corresponding results show that the brown-out detection responds to supply voltage deviation only if the supply voltage is below the predefined threshold voltage during at least

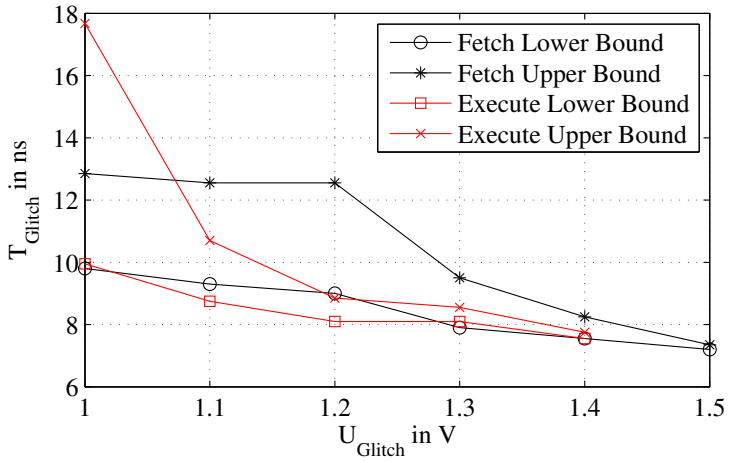

Figure 6.2: Time interval of the clock glitch period T_{Glitch} as a function of the underpowering voltage U_{Glitch} for successfully attacking the fetch and the execution stage of the Cortex-M0.

two consecutive rising edges of the clock signal. In contrast to the brown-out interrupt, the brown-out reset was correctly performed for all $U_{Glitch} < U_{th,rst}$.

The presented results for the attacks on the instruction execution procedure of the Cortex-M0 show that the fetch stage and the execution stage are influenced by clock glitch attacks. Therefore, we additionally analyzed the relationship between the underpowering voltage U_{Glitch} and the clock glitch period T_{Glitch} either attacking the fetch stage or the execution stage. For each stage, Figure 6.2 depicts the range of T_{Glitch} for successfully inducing a fault as a function of the applied underpowering voltage level U_{Glitch}. The presented results are obtained by attacking the `adds Rd,Rn` instruction. Regardless of the attacked stage, the lower and upper bounds of T_{Glitch} increase with lower values for U_{Glitch}. Furthermore, the interval of T_{Glitch} increases with a lower underpowering voltage U_{Glitch}. These effects confirm theoretical assumptions about the relation between the supply voltage level and the resulting timing behavior of combinational logic. At an underpowering voltage of $U_{Glitch} = 1.2\,\mathrm{V}$, the intervals of T_{Glitch} for affecting the fetch and the execution stage do not overlap. Based on this fact, it is possible to attack either the fetch or the execution stage by properly selecting the applied clock glitch period T_{Glitch}. For all attacks performed on the instruction execution procedure of the Cortex-M0 (in our practical experiments), underpowering with $U_{Glitch} = 1.2\,\mathrm{V}$ is applied in addition to the inserted clock glitch. Moreover, the brown-out detection feature is deactivated.

6.1.2 Arithmetical/Logical Instructions

To investigate the effects of clock glitch attacks for the group of *arithmetical/logical* instructions on the Cortex-M0, we attacked the `adds`, the `muls`, and the `lsls` instruction as listed in Table 5.1. All attacks are performed by separately inserting a single clock glitch in each pipeline during the sequence of processing an instruction. In order to retrieve meaningful results, injected faults are separately analyzed for each pipeline stage. Thus, several attack parameters have to be considered. First, the number of clock cycles between the trigger event and the clock glitch insertion defines the attacked pipeline stage. In this context, the position of the attacked instruction within the program code influences

the assigned position within the pipeline due to the fact that two 16-bit instructions are fetched in one cycle. Referring to the pipeline structure of the Cortex-M0 depicted in Figure 4.3, an instruction stays in the pipeline for either three or four consecutive clock cycles. Second, the clock glitch period T_{Glitch} influences the result of an injected fault, depending on the attacked state of the combinational logic. For each investigated instruction, each pipeline stage is analyzed by applying clock glitch attacks with different values for the clock glitch period T_{Glitch}.

We first attacked the `adds Rd,Rn` instruction. The two registers `Rd` and `Rn` are initialized with known values. Within the inline assembly segement, the attacked `adds` instruction is surrounded by `nop` instructions in order to separate the attacked instruction from register initialization and result communication after an attack. The sequence of executed instructions, including the corresponding position within the pipeline of the Cortex-M0, is illustrated in Table 6.1. Regardless of the applied glitch period T_{Glitch}, the decode stage was not susceptible to fault injection. For that reason, we focus on the results achieved by attacking the fetch and the execution stage for the `adds` instruction.

Table 6.1: Attack of a single `adds Rd,Rn` instructions and the corresponding position within the pipeline of the Cortex-M0.

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. Targeting the fetch stage of the instruction `adds Rd,Rn`, a clock glitch is inserted in clock cylce *i+2*, referring to Table 6.1. Based on the resulting value of register `Rd` which should contain the result of the addition, the `adds` is not executed when using a glitch period T_{Glitch} between 9.3 ns and 12.6 ns. Consequently, the value of `Rd` remains unchanged from its initialization value. This behavior can be clarified by considering the next instruction fetch at clock cycle *i+4*. If a clock glitch is inserted during this stage, the `adds Rd,Rn` instruction is executed twice, again determined by the resulting value of `adds Rd`. Repeating the attacks with different initialization values for `Rd` and `Rn` leads to similar effects. By combining both observations, it is possible to state assumptions about the resulting erroneous behavior. Without inserting a clock glitch in cycle *i+4*, the two `nop` instructions are fetched as they are immediately located after the `adds` instruction in the program code. When now inserting a clock glitch in cycle *i+4*, the previous two instructions from the fetch stage in clock cycle *i+2* remain in the instruction fetch buffer and are consecutively decoded and executed. The actual instructions are indirectly replaced by the instructions of the previous fetch stage and are, in further consequence, not executed at all. Similar effects occur when the clock glitch is inserted in cycle *i+2*. Instead of updating the instruction fetch buffer with the `adds` instruction, the `nop` instruction fetched in clock cycle *i* stays in the fetch buffer. In summary, a clock glitch attack on the instruction fetch stage prevents an instruction from being executed by replacing it with the previously fetched instruction.

Execution stage. When attacking the execution stage of the `adds Rd,Rn` by inserting a single clock glitch at cycle *i+5*, different effects occur compared to the attacks performed

on the fetch stage. A glitch period T_{Glitch} between 8.3 ns and 9.2 ns causes wrong results in the destination register *Rd* of the addition `adds Rd,Rn`. The resulting erroneous values vary depending on the applied clock glitch period T_{Glitch} and the addends provided by the values of *Rd* and *Rn*. When increasing T_{Glitch} between 8.3 ns and 9.2 ns, the value of *Rd* is increased too. Repeatedly performed experiments with the same initialization values of *Rd* and *Rn* yield similar erroneous results, only determined by the applied T_{Glitch}. However, no relations between the faulty values and other register entries are observed. The wrong results when attacking the execution stage of `adds Rd,Rn` are assumed to be caused by timing violations of the combinational logic in case of inserted clock glitches. Depending on the applied glitch period T_{Glitch}, different results might be generated by the underlying hardware implementation of the adder and be written to the destination register *Rd*.

In order to provide a more detailed evaluation of the influence of clock glitch attacks on the `adds` instruction, an additional test scenario is used, targeting eight consecutively executed `adds Rd,#imm`. Using this instruction, a constant 8-bit value *#imm* is added to the value of the register *Rd*. After executing the addition, the result equals again the value of *Rd*. The sequence of executed instructions including the corresponding position within the pipeline of the Cortex-M0 is illustrated in Table 6.2. The used registers *Rd* and *Re* are initialized to zero at the beginning. To identify the affected instruction after a fault was injected, the constant values are chosen in a way, that the resulting erroneous values of *Rd* and *Re* reflect, whether an instruction was skipped or executed twice.

Table 6.2: Attack on a set of `adds Rd,#imm` and `adds Re,#imm` instructions including the corresponding position within the piepline of the Cortex-M0.

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 stage. When inserting a single clock glitch with a glitch period T_{Glitch} between 8.3 ns and 9.2 ns during the fetch stage of the instructions `adds Re,#1` and `adds Re,#4` in clock cycle *i+4*, the two instructions previously fetched in cycle *i+2* remain in the fetch buffer. Consequently, the instructions `adds Re,#1` and `adds Re,#4` are replaced by `adds Rd,#16` and `adds Rd,#64`. Subsequently, `adds Rd,#16` and `adds Rd,#64` are executed in clock cycle *i+6* or in clock cycle *i+7*, respectively. Similar behavior is obtained when attacking the fetch stage in cycle *i+2* or in cycle *i+6*. It is notable that four instructions influence the erroneous result when attacking one fetch stage as two of them are skipped and two are executed twice.

Execution stage. As shown in Table 6.2, one instruction gets executed in each clock cycle. Therefore, only one instruction is affected by a single clock glitch inserted in the execution stage. However, attacking the execution stage implies an attack on the fetch stage in every second clock cycle. For example, if the execution of `adds Rd,#16` is attacked in clock cycle *i+4*, the fetch operations for `adds Re,#1` and `adds Re,#4` are additionally affected by this attack. As discussed in Section 6.1.1, the underpowering

voltage of $U_{Glitch} = 1.2\,\text{V}$ allows to separate T_{Glitch} for either attacking the fetch stage or the execution stage. Confirmed by the obtained results, the fetch stage is affected during a glitch period T_{Glitch} between 8.3 ns and 9.2 ns. Influencing the execution stage requires a glitch period T_{Glitch} between 9.3 ns and 12.6 ns. In contrast to that, inserting a clock glitch in cycle *i+2* only affects the execution of `adds Rd,#4`, depending on the used glitch period T_{Glitch}. Again, the decoding stage is not susceptible to the clock glitches inserted during the attacks.

For the group of *arithmetical/logical* instructions, we additionally analyzed the influence of clock glitch attacks on the fetch and execution stage of `muls Rd,Rn` and `lsls Rd,#imm`. Similar to the test scenario of the `adds` instruction, the used registers, `Rd` and `Rn` are initialized within the inline assembly segment of the corresponding test application for the Cortex-M0. For both, the `muls` and `lsls` instruction, we obtained equal results compared to the previously analyzed `adds` instruction when attacking the fetch stage. In this case, the fetch buffer is prevented from being updated with the new instructions from the program memory and the previously fetched instructions remain in the fetch buffer. The glitch period T_{Glitch} for inducing faults lies between 9.3 ns and 12.6 ns once more. In contrast to that, different effects are observed when attacking the execution stage of either the `muls Rd,Rn` or the `lsls Rd,#imm` instructions. In case of attacking the execution stage of `lsls`, the value of register `Rd` is set to zero for a glitch period T_{Glitch} between 8.3 ns and 10.7 ns. This effect occurs independent from the initialization value of `Rd` and the number of left shifts defined by *#imm*. In order to influence the execution of the `muls` instruction, a glitch period T_{Glitch} between 10.2 ns and 20.7 ns has to be applied. As a result, the register `Rd` is set to a variation of different values. As already observed during the attacks of the execution stage of `adds`, the erroneous value of `Rd` increases with a higher glitch period T_{Glitch}. Although the erroneous values depend on the initially assigned values of `Rd` and `Rn` as well as on the applied clock glitch period T_{Glitch}, relations to other register entries are not statable. The reasons for this behavior are assumed to be caused by timing violations based on the inserted clock glitch. Therefore, the underlying hardware multiplier of the Cortex-M0 is influenced in different intermediate states during the calculation of the result depending on the applied glitch period.

6.1.3 Branch Instructions

For the group of *branch* instructions, we analyzed the effects of clock glitch attacks on the Cortex-M0 by attacking the `beq label` instruction. The influences of injected faults on processing branch instructions are separately analyzed for each pipeline stage with different values for the clock glitch period T_{Glitch}. Therefore, a single clock glitch is inserted in each attack run where the delay between the trigger event and the analyzed instruction is chosen according to the attacked pipeline stage. Again, the pipeline structure of the Cortex-M0 has to be considered in our practical experiments, particularly focusing on the fetch stage, when two 16-bit instructions are loaded from program memory during one clock cycle. Furthermore, an additional instruction has to be executed immediately before `beq label` in order to define the condition for the branch. In our test application, `cmp Rd,Rn` is used to set the zero flag of the application program status register which is further accessed by the `beq` instruction. Within the inline assembly code segment of the test application, several operations are performed to provide meaningful results for evaluating the influence of injected faults. First, the two registers `Rd` and `Rn` are initialized to equal values at the beginning. In doing so, the condition for performing the branch with the `cmp Rd,Rn`

instruction is satisfied. Second, the `cmp` instruction immediately followed by the attacked `beq` instruction are subsequently placed after the initialization of the registers. Before and after these two instructions, several `nop` instructions are inserted to prevent any side effects or interferences with other code parts during the attack. The attacked part of the assembly code segment is illustrated in Table 6.3, including the corresponding positions of the instructions within the pipeline of the Cortex-M0. The third part of the assembly code segment consists of the destination address for the branch instruction, defined by the *label*. Known values are added to a previously initialized register before and after the *label* in order to identify if the branch was carried out or not. As already observed during the attacks of the `adds` instruction, the decode stage was not susceptible to fault injection in case of attacking the `beq` instruction.

Table 6.3: Attack of the `beq label` instructions and the corresponding position within the pipeline of the Cortex-M0.

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				
`cmp Rd,Rn`			F	D	E			
`beq label`			F		D	E		
`nop`					F	D	E	
`nop`					F		D	E

Fetch stage. In normal condition, without inserting a clock glitch, the branch is taken as the previously compared registers `Rd` and `Rn` are initialized to equal values. When inserting a clock glitch during the fetch stage of `beq label` in clock cycle *i+2*, we obtained the same behavior as for attacking the fetch stage of the `adds` instruction. The two `nop` instructions fetched in clock cycle *i* remain in the instruction fetch buffer. Consequently, the `nop` instructions are executed twice instead of executing the `cmp` and `beq` instruction. Therefore, it is possible to prevent a branch and to further manipulate the program flow. The clock glitch period T_{Glitch} for successfully inducing this fault in the fetch stage is between 9.3 ns and 12.6 ns. As depicted in Table 6.3, both instructions are positioned within the pipeline in order to be fetched in one clock cycle. However, similar behavior is observed, if the two instructions are fetched in different clock cycles and only one of the corresponding fetch stages is attacked. In this case, either the compare or the branch operation are not performed.

Execution stage. By inserting a clock glitch during the execution stage of the `beq label` instruction in clock cycle *i+5*, we were not able to achieve any impact on the behavior of the instruction. Independent from the applied clock glitch period T_{Glitch}, the branch instruction was correctly executed and the program flow continued at the position defined by the *label*.

6.1.4 Memory Instructions

For the group of *memory* instructions of the Cortex-M0, we considered `ldr Rd,[Rn]` and `str Rd,[Rn]` in our practical experiments. The influence of injected faults on processing of instructions are separately analyzed for each pipeline stage, taking into account that an additional clock cycle for execution is required by both instructions. Based on the configurable delay between the trigger event and the analyzed instruction, a single clock

glitch is inserted to a specific pipeline stage. Moreover, different values for the clock glitch period T_{Glitch} are used during the attacks. In order to reproduce the resulting erroneous behavior after an injected fault, several operations are performed by the used test application. A known value is stored to a specific address of the SRAM. The address is further used by `ldr` and `str` to access the memory. The inline assembly code segment contains the attacked load or store instruction. Additionally, the registers `Rd`, `Rn` are initialized at the beginning of this segment. Therefore, `Rd` is set to a nonzero number and the previously specified memory address is written to `Rn`. After the attacked instruction, the SRAM is again accessed by an additional load instruction in order to read back the written value. To prevent any interferences with other code parts, `nop` instructions are inserted before and after the attacked instruction. In the following, the obtained results for fetch stage and execution stage are presented for both investigated memory instructions, `ldr Rd,[Rn]` and `str Rd,[Rn]`. However, attacks on the decode stage were ineffective again.

Fetch stage. In contrast to *arithmetical/logical* and *branch* instructions where the fetch buffer was prevented from being updated in case of inserting a clock glitch in the fetch stage, deviations in the resulting effects for the `ldr` and `str` instructions are observed. When inserting a clock glitch in the fetch stage of the `ldr Rd,[Rn]` instruction, the value of `Rd` was set to zero instead of loading the value stored in memory at the address given by the register `Rn`. Similar effects were observed for the `str Rd,[Rn]` instruction. In case of attacking the fetch stage of this instruction, the stored value was zero instead of the value in register `Rd`. Applicable to both instructions, a glitch period T_{Glitch} between 10.2 ns and 10.9 ns led to this behavior.

Execution stage. Both investigated *memory* instructions require two clock cycles for execution. Results show that only the second execution stage of either `ldr Rd,[Rn]` or `str Rd,[Rn]` was vulnerable to an inserted clock glitch. In case of the `str` instruction we observed that the SRAM address, stored in register `Rn` is written to the memory at the given memory address instead of the value given by `Rd`. In contrast to that, the SRAM address is loaded to `Rd` instead of the value stored in memory at the memory address given by `Rn` when attacking the execution stage of the `ldr` instruction. In case of the `ldr`, the resulting erroneous behavior corresponds to the operation of a register transfer instruction, i.e. *mov Rd,Rn*. Both phenomena occur when applying a glitch period T_{Glitch} between 9.0 ns and 9.6 ns. For different values of T_{Glitch} between 8.3 ns and 8.9 ns, neither `ldr Rd,[Rn]` or `str Rd,[Rn]` are executed.

6.2 ATxmega 256

The attacks on the instruction execution procedure of the ATxmega 256 are also performed by using the common attack scenario presented in Section 5.3. After each attack, the values of the 8-bit general purpose registers $R0$ to $R31$ are transferred to the control computer for further analysis of the induced faults. As shown in Figure 4.2, the pipeline of ATxmega 256 consists of two stages, namely the fetch stage and execution stage. In each clock cycle, a 16-bit instruction is loaded from the program memory while the previously fetched instruction is executed simultaneously. It turned out that, in contrast to the Cortex-M0, the ATxmega 256 is more sensitive to clock glitch attacks and underpowering was not necessary for successful fault injection. However, the ATxmega 256 provides a brown-out detection mechanism with two modes of operation which are continuous and sampled BOD. As stated in the datasheet [6], the brown-out detection time t_{BOD} is 1 ms

in sampled mode and 400 ns in continuous mode. Due to the fact that clock glitch attacks without additional underpowering are sufficient for successfully inducing faults during the instruction execution procedure of the ATxmega 256 and the time intervals of the brown-out detection are far above the time we would require for short-time underpowering, the brown-out detection was not further investigated. The presented results for the ATxmega 256 are obtained by inserting clock glitches without using underpowering, meaning that the supply voltage is left unchanged at the nominal voltage level of 3.3 V during an attack.

6.2.1 Arithmetical/Logical Instructions

The investigated instructions for the group of *arithmetical/logical* instructions on the ATxmega 256 include the addition of two registers values, using `add Rd,Rn` and the multiplication of two register values using `mul Rd,Rn`. The effects of clock glitch attacks on the instruction execution procedure of these two instructions are analyzed individually for the fetch stage and the execution stage. In contrast to the Cortex-M0, in each clock cycle one 16-bit instruction is fetched while the previously fetched instruction is executed. Therefore, two instructions are affected by a single clock glitch depending on the applied clock glitch period T_{Glitch}. The following attack parameters have to be considered for attacks on the instruction set of the ATxmega 256. First, the attacked stage of an instruction, either fetch or execute, is determined by the number of clock cycles between the trigger event for the fault board and the clock glitch insertion. Second, different erroneous results can be expected, depending on the clock glitch period T_{Glitch}.

The first instruction investigated in our practical experiments on the ATxmega 256 is the `add Rd,Rn` instruction. In the used test application, the attacked instruction is placed within the inline assembly code, including the initialization of the two 8-bit registers `Rd` and `Rn`. In order to separate initialization and result communication from the attacked instruction, `nop` instructions are inserted before and after `add Rd,Rn`. Table 6.4 illustrates the sequence of executed instructions during the attack including the positions within the pipeline.

Table 6.4: Attack of a single `add Rd,Rn` instruction and the corresponding position within the pipeline of the ATxmega 256.

Instruction	**Cycle**					
	i	*i+1*	*i+2*	*i+3*	*i+4*	*i+5*
`nop`	F	E				
`nop`		F	E			
`add Rd,Rn`			F	E		
`nop`				F	E	
`nop`					F	E

Fetch stage. When attacking `add Rd,Rn` by inserting a clock glitch in the fetch stage in clock cycle *i+2*, the resulting effects are similar to those already observed on the Cortex-M0. Instead of performing the addition of the two values of register `Rd` and `Rn`, the initialization value is still present in the destination register `Rd` after the attack. This behavior can also be explained by the assumption that the fetch buffer is not updated with the actual `add` instruction and the `nop`, previously fetched in clock cycle *i+1*, remains in the fetch buffer. A glitch period T_{Glitch} between 5.9 ns and 17.9 ns led to this result. By inserting a clock glitch in the fetch stage of the `nop` instruction in cycle *i+3*, the

`add` instruction remains in the fetch buffer and is consequently executed twice. The only difference to the attack of the fetch operation in cycle *i+2* is given by the fact that another range of T_{Glitch} between 9.3 ns and 15.3 ns is required. Different timing behavior for updating the fetch buffer might lead to this variation of the time interval for T_{Glitch}, depending on the transition between the previous and the following value of the fetch buffer.

Execution stage. Targeting the execution stage of `add Rd,Rn`, a clock glitch inserted in cycle *i+3* modifies the value of the destination register `Rd`. For a glitch period T_{Glitch} between 5.9 ns and 7.2 ns, `Rd` is changed to recurring wrong values without explainable relations to other register or memory entries.

Considering now `mul Rd,Rn`, which is the second investigated instruction from the group of *arithmetical/logical* instructions on the ATxmega 256, particularly interesting differences are given compared to the `add` instruction. First, the 16-bit result of multiplying `Rd` and `Rn` is divided into low and high byte and written to the 8-bit general purpose registers `R0` and `R1`. Second, the execution of `mul Rd,Rn` requires two clock cycles. Two clock cycles have to be considered when attacking the execution stage of this instruction. The attacked `mul` instruction is again placed between `nop` instructions within the inline assembly code of the test application.

Fetch stage. When attacking the fetch stage of `mul Rd,Rn`, the instruction is not executed using a glitch period T_{Glitch} between 5.9 ns and 17.9 ns. This behavior complies with the previously obtained results for the `add` instruction where the instruction fetch buffer is not updated in case of inserting a clock glitch during the fetch operation. Moreover, the same time interval for the glitch period applies for both instruction. However, we were not able to influence the next fetch stage in one of the following clock cycles, in order to provoke a double-execution of the `mul` instruction.

Execution stage. Inserting a clock glitch in the second execution cycle of the `mul Rd,Rn` instruction led to wrong multiplication results in register `R1` when using a clock glitch period T_{Glitch} between 6.4 ns and 7.8 ns. In contrast to successfully manipulating the high byte of the multiplication result which is stored in register `R1` the correct value for the low byte was written to `R0`. However, no effects were observed when attacking the first execution cycle of the `mul` instruction. In this case, both result bytes were correctly written to the the result registers, regardless of the applied glitch period T_{Glitch} and the initialization values of `Rd` and `Rn`.

6.2.2 Branch Instructions

For the group of *branch* instructions, we analyzed the behavior of the conditional branch `breq` *`label`*, similar to the one investigated for the Cortex-M0. Both pipeline stages are individually attacked by either inserting a clock glitch during the fetch stage or the execution stage of the instruction. In order to fulfill the condition for taking the branch, `cp Rd,Rn` is executed straight before the attacked branch instruction. The two registers `Rd` and `Rn` are initialized with the same values, further setting the zero flag of the status register after the `cp Rd,Rn` is executed. The destination address for the branch is defined by *label* within the assembly code segment. Thus, the test scenario for the attack of the `breq` *`label`* instructions equals the procedure already used for the Cortex-M0. Table 6.5 shows the attacked part of the assembly code segment with the corresponding positions within the pipeline stages of the ATxmega 256.

Fetch stage. The fetch operation of `breq` *`label`* is attacked by inserting a clock glitch

Table 6.5: Attack of a single `breq label` instructions and the corresponding position within the pipeline of the ATxmega 256.

Instruction	Cycle					
	i	*i+1*	*i+2*	*i+3*	*i+4*	*i+5*
`nop`	F	E				
`cp Rd,Rn`		F	E			
`breq label`			F	E		
`nop`				F	E	
`nop`					F	E

in cycle *i+2*. As a result, the branch is not taken when using a glitch period T_{Glitch} between 6.7 ns and 18.2 ns. Based on the effects observed in previous attacks of fetch stage, the `cp Rd,Rn` instruction, fetched in clock cycle *i+1* can be assumed to remain in the instruction fetch buffer. Consequently, instead of executing the branch instruction, the `cp` instruction is executed twice, which in turn allows a manipulation of the program flow without directly influencing data. Moreover, executing `cp Rd,Rn` twice has only an impact on the following instruction if the value of zero flag is crucial for this instruction. Additionally, the obtained behavior complies with the effects already obtained for attacking the fetch stage of the *arithmetical/logical* instructions.

Execution stage. The execution of the branch instruction `breq label` was not vulnerable to the performed clock glitch attacks. When inserting a clock glitch in cycle *i+3*, the branch is correctly taken, independent from the applied clock glitch period. Again, this behavior matches the results obtained for the branch instruction of the Cortex-M0.

6.2.3 Memory Instructions

To investigate the effects of clock glitch attacks for the group of *memory* instructions on the ATxmega 256, we attacked the `ld Rd,X` instruction and the `st X,Rn` instruction. For each instruction, the fetch and the execution stage are attacked by inserting a single clock glitch. In case of using `ld Rd,X` to load an 8-bit value from SRAM, one additional clock cycle is required. Again, different values for the glitch period T_{Glitch} are used in order to cover the entire range of possible effects on the execution procedure for the `ld` and the `st` instruction. Within the test application for the ATxmega 256 a known value is written to a specific memory address of the internal SRAM. The 16-bit address pointer `X` is initialized at the beginning of the inline assembly code segment by loading the previously specified memory address. Additionally, the registers `Rd` and `Rn` are initialized with nonzero numbers. After that, several `nop` instructions are placed before and after the attacked *memory* instruction in order to prevent any influence of other instructions to the resulting erroneous behavior. In case of the `st` instruction, the memory address is accessed once again at the end of the assembly code segment in order to compare the actual value after an attack with the correct one. In the following, the obtained results for the `ld Rd,X` the `st X,Rn` instruction are presented.

Fetch stage. As already obtained in our previous experiments concerning the fetch stage of the ATxmega 256, the instructions `ld Rd,X` and `st X,Rn` are not executed, when inserting a clock glitch during the fetch operation. A glitch period T_{Glitch} between 5.9 ns and 17.9 ns provokes this behavior for both instructions. Again, the previous instruction remains in the fetch buffer and is executed twice instead of the load or store instruction. Thus, it is possible to manipulate the data flow by preventing data from being stored in

memory or loaded to a register.

Execution stage. Attacking the `st X,Rn` instruction in the execution stage results in wrong data written to the memory address given by the address pointer `X`. In case of inducing this fault, correct memory addressing can be assumed due to the fact, that only the memory location, defined by `X` is affected. Similar effects are observed when `ld Rd,X` is attacked in the first clock cycle of execution. Instead of loading the correct value from memory, wrong data was loaded to `Rd` after an attack. However, no relation between the resulting wrong value of `Rd` and any value at a different memory location can be stated. For both instructions, a glitch period T_{Glitch} between 8.1 ns and 8.8 ns led to this behavior. Additional effects are observed, when inserting a clock glitch with T_{Glitch} between 9.6 ns and 10.3 ns in the first execution cycle of `ld Rd,X`. In this case, the value of `Rd` was zero, regardless of the stored value at the given memory address. In contrast to the load instruction, the execution of `st X,Rn` was invulnerable for all other values of T_{Glitch}.

6.3 Discussion and Comparison

The obtained results show that clock glitch attacks can be used to influence the processing of instructions on a microcontroller. Both investigated MCUs, the Atmel ATxmega 256 and the ARM Cortex-M0, are vulnerable to these kind of attacks. Due to the reproducible behavior of the identified faults, it is possible to define reasonable fault models for specific software implementations. Particularly for cryptographic primitives or any other security-relevant application, the effects of clock glitch attacks have to be necessarily considered during software development. Illustrated in Table 6.6 for the Cortex-M0 and in Table 6.7 for the ATxmega 256, the obtained results and effects are summarized. The fetch stage exposed to be reliable in terms of the resulting erroneous behavior. The memory instructions of the Cortex-M0 represent the only exception. All other investigated instructions of the Cortex-M0 and the ATxmega 256 show a similar behavior when inserting a clock glitch during the fetch operation. Instead of loading the actual instruction from program memory to the instruction fetch buffer, the previously loaded instruction remains in the buffer and is consequently executed twice. Due to the additionally inserted rising edge in the clock signal, the resulting shortened time interval is insufficient for updating the instruction fetch buffer and the process of fetching a new instruction is undercut. A modification of the program counter can be excluded. Otherwise, an instruction is either executed twice if the program counter is not incremented or skipped if the program counter gets incremented twice. A considerable difference between the Cortex-M0 and the ATxmega 256 is given by the pipeline structures of the MCUs. When attacking the fetch stage of the Cortex-M0, two 16-bit instructions can be influenced by a single clock glitch. As a result, four instructions are responsible for the corresponding erroneous result after a single attack of the fetch stage in which the two instructions previously fetched are executed twice instead of two actual instructions. For successfully inducing this behavior, the same interval for T_{Glitch} applies for all investigated instructions of the Cortex-M0, apart from memory instructions. Compared to this behavior, the fetch operation of a single 16-bit instruction can be influenced in each clock cycle on the ATxmega 256. Thus, two instructions are involved in a single attack of the fetch stage and further determine the corresponding faulty result. Only the investigated branch instruction slightly differs from the generally observed interval for T_{Glitch}. In this case, the *cp* instruction replaces the branch instructions when inserting a clock glitch during the fetch operation of the

Table 6.6: Summary of the results for the investigated instructions of the Cortex-M0.

Instruction	Pipeline Stage	T_{Glitch} ns	Effects
`adds Rd,Rn`	Fetch	9.3 to 12.6	`adds` not executed
	Execute	8.3 to 9.2	wrong results in *Rd*
`muls Rd,Rn`	Fetch	9.3 to 12.6	`muls` not executed
	Execute	10.2 to 10.7	wrong results in *Rd*
`lsls Rd,#imm`	Fetch	9.3 to 12.6	`lsls` not executed
	Execute	8.3 to 10.7	*Rd* set to zero
`beq label`	Fetch	9.3 to 12.6	`beq label` not executed
	Execute	-	no effects observed
`ldr Rd,[Rn]`	Fetch	10.2 to 10.9	*Rd* set to zero
	Execute	8.3 to 8.9	`ldr Rd,[Rn]` not executed
		9.0 to 9.6	address written to *Rd*
`str Rd,[Rn]`	Fetch	10.2 to 10.9	memory entry set to zero
	Execute	8.3 to 8.9	`str Rd,[Rn]` not executed
		9.0 to 9.6	address written to memory entry

branch instruction. All other investigated instructions are replaced by a *nop*. A reasonable explanation for this behavior might be that the time interval for the transition between the previous and the actual instruction varies, depending on the opcode of the instructions when updating the fetch buffer. In general, skipping a specific instruction provides a powerful opportunity for an attacker since almost any modification of data and program flow is possible. In case of software implementations of cryptographic primitives, skipping arithmetical or logical instructions might leak information about internal intermediate states of cryptographic calculations. As a result, it might be possible to retrieve secret information or other details about the implementation which can be further used to reveal supposedly secure data. For example, skipping the addition of the final round key during AES encryption allows to calculate the AES key if a pair of ciphertexts can be generated from the same plaintext, one without performing the last round key addition. In case of branch instructions, both MCUs, the Cortex-M0 and the ATxmega 256, showed a vulnerability to clock glitch attacks during the fetch operation. Therefore, it is possible to prevent entire code segments or loop iterations from being executed when skipping a single branch instruction. As a consequence, leakage of secret information is again feasible, based on program flow modifications. Side effects caused by the double-execution of the previous instruction are minimized by the fact that branch instructions are usually executed directly after compare instructions. When executing the compare instruction twice instead of the branch, the CPU status flags are changed for the first instruction executed after the attack. Additional modification of data or program flow is only possible if this instruction is sensitive to one of the status flags. In summary, our practical attacks of the fetch stage revealed a considerable weakness on both MCUs. Particularly the wide range of T_{Glitch} applicable in order to achieve the behavior of skipping instructions provides a robust fault model.

Table 6.7: Summary of the results for the investigated instructions of the ATxmega 256.

Instruction	Pipeline Stage	T_{Glitch} ns	Effects
`add Rd,Rn`	Fetch	5.9 to 17.9	`add` not executed
	Execute	5.9 to 7.2	wrong results in *Rd*
`mul Rd,Rn`	Fetch	5.9 to 17.9	`add` not executed
	Execute	6.4 to 7.8	wrong results in *Rd*
`breq` *label*	Fetch	6.7 to 18.2	`breq` *label* not executed
	Execute	-	no effects observed
`ld Rd,X`	Fetch	5.9 to 17.9	`ld Rd,X` not executed
	Execute	8.1 to 8.9	wrong results in *Rd*
		9.6 to 10.3	*Rd* set to zero
`st X,Rn`	Fetch	5.9 to 17.9	`ld Rd,X` not executed
	Execute	8.1 to 8.9	wrong value written to memory entry

In contrast to the obtained results for the fetch stage, different behavior can be observed when inserting the clock glitch during the execution stage of an instruction. In case of the investigated arithmetical and logical instructions, results show that faults can be induced during the execution stage, leading to wrong calculation results in the destination register. Similar behavior can be observed for load and store operations on the ATxmega 256 in which wrong values can either be loaded from or written to memory. Beside inducing fault resulting in elusive wrong values, we were able to induce constant faults by setting register values or memory entries to specific constant values. For example, these values are zero or the memory address in case of load and store operations on the Cortex-M0, depending on the applied glitch period. Again, the results of the attacks on the execution stage reveal a thread on cryptographic software implementations for both MCUs. Particularly the possibility of setting register or memory entries to specific known values during cryptographic calculations might be used to retrieve secret information. Using the example of key addition during AES encryption, the final round key might be accessible in the plaintext if the AES state bytes can be set to a constant known value in a previous operation during encryption before the final round key is added.

So far, the presented results almost disregard the simultaneous influence of a single clock glitch on several pipeline stages. In our test applications, *nop* instructions are used in order to prevent any effects of other instructions on the attack results of the investigated instruction. However, several instructions are affected by a single clock glitch, when attacking a real software implementation of a cryptographic primitive. In this case, several aspects have to be considered. As illustrated in Figure 4.3 for the Cortex-M0, the fetch stage is only affected in every second clock cycle, whereas the execution of an instruction can be influenced in every clock cycle. For the ATxmega 256, two instructions are affected in every clock cycle, one in the execution and one in the fetch stage, as shown in Figure 4.2. A properly chosen value for the glitch period T_{Glitch} helps in some cases to prevent an influence of several stages under consideration of the involved instructions. Additionally, the underpowering voltage level can be used to change the interval of T_{Glitch}, as discussed for the Cortex-M0 in Section 6.1.1.

Chapter 7

ATxmega 256 AES Crypto Engine

In 2001, the *National Instituted of Standards and Technology (NIST)* announced the *Advanced Encryption Standard (AES)* based on the *Rijndael* cipher as successor to the *Data Encryption Standard (DES)*. Today, AES is widely established for symmetric-key cryptography and is used for a huge number of security-relevant hardware and software applications. Due to the fact that the AES algorithm can be efficiently implemented in hardware, security applications can benefit from an abundant availability of hardware-supported cryptographic acceleration. A variety of microcontrollers also support this feature in terms of cryptographic co-processing. In this context, the key used for encryption and decryption is typically stored within the device and is never directly exposed through an interface of the device. However, fault attacks aim at revealing such secret information by provoking undefined operation conditions or injecting faults specifically during cryptographic calculations.

This chapter focuses on a fundamental description of the AES algorithm providing introductory basics about the encryption procedure further relevant for the attacks presented in Chapter 8. Additionally, the AES crypto engine of the ATxmega 256 is introduced with its properties and features, followed by a description of the attack scenario used for our experiments.

7.1 Advanced Encryption Standard (AES)

The *Advanced Encryption Standard (AES)* is a symmetric-key block cipher. Symmetric ciphers use the same key for both the encryption and the decryption procedure of a message. Block ciphers operate on fixed-length data blocks. In this context, AES uses a fixed block length of 128 bits and supports three different key lengths of 128 bits, 192 bits, or 256 bits. The AES algorithm consists of repeatedly executed round operations performed on a matrix representation of four-by-four bytes called *state*. Figure 7.1 illustrates how the 16 state bytes $a_0, a_1, \ldots, a_{15}$ are arranged. The key length used for encryption or decryption specifies the number of executed round iterations.

- A 128-bit key implies 10 round iterations.
- A 192-bit key implies 12 round iterations.
- A 256-bit key implies 14 round iterations.

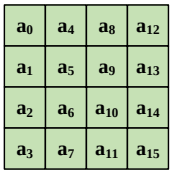

Figure 7.1: Matrix representation of the AES state.

In the following description we focus on the AES version using 128-bit keys as this is the only supported key length of the evaluated AES crypto engine on the ATxmega 256. For more detailed information about the AES we refer to Paar and Pelzl [32] providing an enlighten description of the AES and, of course, to the book [11] written by Rijmen and Daemen, the two designer of *Rijndael*. Additionally, the official AES specification can be found at [29].

Each of the aforementioned rounds, except for the last, consists of four transformations: *SubstituteBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. In order to provide a similar encryption and decryption sequence, the last *MixColumns* transformation is omitted. Figure 7.2 illustrates the encryption process based on the 128-bit version. The ciphertext $c = AES_k(p)$ is generated by applying the AES algorithm to the plaintext p using the key k. In addition to the repeatedly performed round operations, the state is initialized at the beginning of the encryption procedure. In doing so, *AddRoundKey* is used once to combine the plaintext with the initial key. Subsequently, the round operations are performed. For several processing steps within the round operations, finite field arithmetic in the Galois-field $GF(2^8)$ is used. Arithmetic operations in a finite field are performed on a limited set of elements where the result of an operation is again one of these elements. As the background of finite field arithmetic is not necessarily required for an introductory description of the AES, we refer to [32] for further information about AES-related Galois-field arithmetic.

7.1.1 *SubstituteBytes*

Confusion is one property used in cryptography to obscure the relationship between the ciphertext and the key by making the statistical correlation as complex as possible. The *SubstituteBytes* provides this property based on a nonlinear transformation performed on the state. Each byte a_i of the state is substituted by another byte b_i where the *Rijndeal* substitution box (S-Box) is used for the transformation of a given input value a_i, to a specific output value $b_i = S(a_i)$. For any possible 8-bit input value, the S-Box provides a bijective mapping to a corresponding 8-bit output value where the reverse process is necessary for decryption. Mathematically, the S-Box is based on computing the multiplicative inverse of a number in the Galois-field $GF(2^8)$ and on applying an affine transformation afterwards. However, for most hardware and software implementations of AES the S-Box is usually realized as lookup table without implementing the mathematical construct of it. For more detailed information and mathematical descriptions of the S-Box we again refer to [11, 29, 32].

7.1.2 *ShiftRows*

Diffusion is another property used in cryptography to ensure that statistical structures of the plaintext can't be found in the statistics of the ciphertext. Thus, changing a single

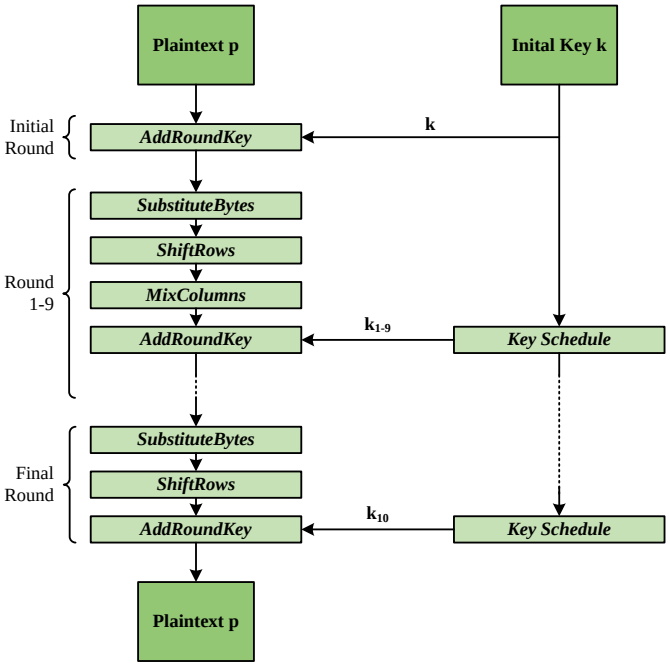

Figure 7.2: AES encryption procedure with a key size of 128 bits.

plaintext bit should affect many ciphtertext bits, widespread in the state. Diffusion on a byte level is therefore given by a permutation using *ShiftRows*. As the name implies, this transformation cyclically shifts the bytes of the state within a row. The *ShiftRows* transformation is performed on each row where a specific offset is applied depending on the row. The first row stays unchanged. The second row of the state is cyclically shifted by one byte to the left, the third row by two bytes, and fourth row by three bytes. Figure 7.3 illustrates the *ShiftRows* transformation based on the AES state representation.

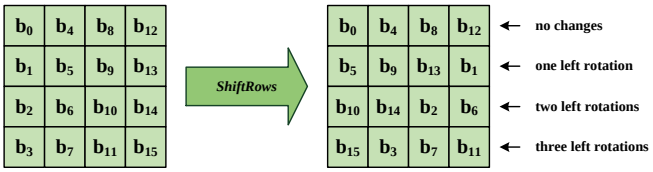

Figure 7.3: *ShiftRows* transformation of the AES state.

7.1.3 *MixColumns*

In addition to the *ShiftRows* transformation, the *MixColumns* operation is the second part of the diffusion step, mixing the four bytes in each column of the state. Concerning the diffusion property, the value of each byte within a column influences the entire column. The *MixColumns* operation can be understood as a matrix multiplication of each column with a constant four-by-four byte matrix. Figure 7.4 illustrates how the output values of *MixColumns* are calculated for the first column of state. The same constant matrix is used for each column and in each round performing *MixColumns*. The additions and multiplications used for the vector-matrix multiplications are applied in the Galois-field $GF(2^8)$. The addition in $GF(2^8)$ represents an bitwise exclusive OR (XOR) operation.

As shown in Figure 7.4, the constant matrix contains only the hexadecimal values 0x01, 0x02, and 0x03. Thus, the multiplication in $GF(2^8)$ for AES only needs to consider three operations. A multiplication by 0x01 results in no changes as 0x01 is the identity element. Multiplication by 0x02 equals a left shift operation by one bit. By adding the original value to the shifted value, multiplication by 0x03 can be done. Depending on the application, also a lookup-table implementation for the vector-matrix multiplication in $GF(2^8)$ might be possible.

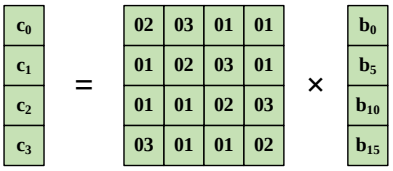

Figure 7.4: *MixColumns* transformation of the first column based on the vector-matrix multiplication of the first column with a constant matrix, further applied to all columns of the state.

7.1.4 *AddRoundKey*

AddRoundKey combines the actual state with a round key. As shown in Figure 7.5, each byte c_i of the state is bitwise XORed to the corresponding byte k_i of the round key, resulting in $d_i = c_i \oplus k_i$. The *AES key schedule* is used to derive the 16-byte round keys from the 16-byte initial key.

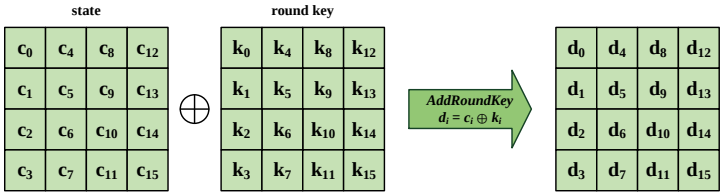

Figure 7.5: The *AddRoundKey* operation adds the round key to the actual state using the XOR operation.

7.1.5 Key Schedule

The *AES key schedule* performs several transformations of the initial key including cyclic shift operations of bytes within a column, substitutions using again the *Rijndeal* S-Box, and additions of round coefficients. The following equations describe how the key bytes $k_{r,b}$ for each round $r = 0, \ldots, 10$ are calculated where $b = 0, \ldots, 15$ indicates an individual key byte in the matrix representation of the key. For the initial round $r = 0$, the initial key k is used without modification: $k_{0,b} = k$. For the other round keys, the first column of the next round key $k_{r,0}$, $k_{r,1}$, $k_{r,2}$, $k_{r,3}$ is given in Equation 7.1. The bytes within the last column of the previous round key are cyclically shifted by one byte and substituted using the S-Box. After this, the bytes within the first column of the previous key are added concluding with an addition of the round coefficient RC_r to the first key byte $k_{r,0}$. All other key bytes $k_{r,b}$ for $b = 4, \ldots, 15$ are then calculated using Equation 7.2. For a detailed

description of the AES key schedule the following references are mentioned: [11, 29, 32].

$$\begin{aligned} k_{r,0} &= k_{r-1,0} \oplus S(k_{r-1,13}) \oplus RC_r \\ k_{r,1} &= k_{r-1,1} \oplus S(k_{r-1,14}) \\ k_{r,2} &= k_{r-1,2} \oplus S(k_{r-1,15}) \\ k_{r,3} &= k_{r-1,3} \oplus S(k_{r-1,12}) \end{aligned} \tag{7.1}$$

$$k_{r,b} = k_{r-1,b} \oplus k_{r,b-4} \qquad (b = 4, \ldots, 15) \tag{7.2}$$

7.1.6 Decryption

Roughly speaking, AES decryption can be performed by applying the encryption procedure in reverse order and using inverse round operations. For the inverse *AddRoundKey*, the XOR operation can be applied again as XOR is its own inverse. It is notable that in case of decryption, *AddRoundKey* starts with the last round key used for encryption, meaning, the round keys are used in inverse order which in turn requires an inverse key schedule. The *MixColumns* operation is inverted by using the inverse constant matrix for multiplication. In the first round of decryption, inverse *MixColumns* is omitted, just as *MixColumns* is not performed in the last round of encryption. Since the AES S-Box provides a bijective mapping, an inverse S-Box can be used for the decryption procedure in order to perform the inverse *SubstituteBytes* operations. In case of the inverse *ShiftRows* operation, the bytes have to be shifted in the opposite direction according to the number of shifts performed during encryption.

7.2 ATxmega 256 AES Crypto Engine

Beside a wide range of peripheral features typically implemented on microcontrollers, the ATxmega 256 also provides an integrated hardware implementation of the Advanced Encryption Standard (AES). Although the ATxmega 256 AES crypto engine exposed to be vulnerable to side-channel analysis attacks [21] leaking the secret key, the existing hardware support for AES facilitates the ATxmega 256 to be used in secure applications. In this chapter, we introduce the characteristics and features of the ATxmega 256 AES crypto engine.

Referring to the user manual [5] of the ATxmega 256 and the AES application note [3], the AES crypto engine supports a key length of 128 bits for encryption or decryption of 128-bit data blocks. The execution of one encryption or decryption procedure requires 375 clock cycles excluding key and state initialization. The AES crypto engine provides an 8-bit state register (`STATE`) and an 8-bit key register (`KEY`) for either writing to or reading from the internal 128-bit state and key memories. To provide sequential access to these registers, a zero-initialized internal 4-bit address pointer is used. Read or write operations automatically increment the appropriate pointer. As stated in [5], the state and key registers can only be accessed when no encryption or decryption operation is in progress. Additionally, direct memory access (DMA) can be used to handle communication with the AES crypto engine. The following step-by-step procedure describes how encryption or decryption is performed.

- At the beginning, the control register (`CTRL`) is used to select encryption or decryption mode. Additionally, several optional features can be configured including the

options of resetting the AES crypto engine, starting encryption/decryption automatically after the state memory is loaded, or performing an XOR operation on the current state entries and the values written to the state register (`STATE`).

- The AES interrupt can be enabled optionally using the interrupt control register (`INTCTRL`).
- The key bytes are sequentially loaded into the key memory through the key register (`KEY`). The initial AES key is used in case of encryption mode. For the decryption mode, the key memory has to be initialized with the last round key used for encryption. This can be either achieved by calculating the last round key in software or by performing a dummy encryption with the initial key as the last round key remains in the key memory after encryption has finished.
- The data bytes are sequentially loaded into the AES state memory via the state register (`STATE`), either using the plaintext for encryption or the ciphertext for decryption.
- If the auto start mode is not enabled in the control register (`CTRL`), the AES crypto engine is started by setting the start flag in the control register (`CTRL`).
- After the encryption/decryption is complete, the ready flag in the AES status register (`STATUS`) is set and an interrupt is generated optionally. The state memory contains the generated ciphertext in case of encryption or the plaintext in case of decryption. The result can be again accessed by sequentially reading the state register (`STATE`).

As already mentioned, the final round key, in case of encryption or the initial key, in case of decryption remains in the key memory. The key memory has to be initialized repeatedly for consecutive encryption or decryption of multiple 128-bit data blocks. As with the state memory, the key memory can be accessed after encryption or decryption by subsequently reading the key register (`KEY`). Consequently, injected faults during the key schedule operation of the AES crypto engine can be identified, improving the capabilities of the fault-based black-box characterization of the AES crypto engine, shown in Chapter 8.1.

7.3 Attack Scenario

As already discussed in Chapter 6, the instruction execution procedure of the ATxmega 256 can be influenced by injecting faults using clock glitch attacks. In this section we describe the attack procedure used for manipulating another particularly interesting feature of the ATxmega 256: the AES crypto engine. To analyze the possible influence of fault attacks on the AES crypto engine, clock glitch attacks are applied during the encryption procedure of the AES crypto engine. The fault injection setup is based on the fault board and the ATxmega 256 extension board described in Chapter 4. An important characteristic of the AES crypto engine has to be considered in order to define an appropriate attack scenario. The key and state memory can only be accessed after the entire encryption procedure has finished. Consequently, the approach of differential fault analysis (DFA) is used, comparing correct and faulty ciphertexts to investigate the impact of clock glitch attacks. In order to obtain two comparable ciphertext pairs, each attack consists of a reference execution without injecting faults and a second execution in which a clock glitch

attack is applied. Additionally, the results may vary depending on the attacked clock cycle of AES encryption as well as on the glitch period T_{Glitch} used for the inserted clock glitch. For attacking the AES crypto engine, a test application on the ATxmega 256 is used to perform communication with the control computer, to synchronize the attack with the fault board, and to execute the attacked AES encryption. The synchronization during an attack between the microcontroller and the fault board is managed by the control computer using UART communication for both the micrcontroller and the fault board. As illustrated in Figure 7.6, the test application and the operations performed by the control computer comply with the following attack procedure:

- **Initialization.** In the initialization phase, the control computer configures the fault board to define the desired attack parameters as given in Section 4.2, including the glitch period T_{Glitch} and the number of clock cycles between the trigger signal is set and the clock glitch is injected. Additionally, the control computer requests the fault board to reset the ATxmega 256. After the reset, the ATxmega 256 starts an initialization routine to configure the system clock for using the clock signal provided by the fault board. The I/O port configuration is set for the trigger output signal and the UART interface is initialized for communication with the control computer. After this procedure, the ATxmega 256 signals this state to the control computer using the UART interface.

- **Synchronization.** The synchronization stage starts by transferring a 128-bit AES key and a 128-bit plaintext block from the control computer to the test application via the UART interface. After the key and the plaintext are received by the microcontroller, the test application initializes the AES crypto engine according to the procedure given in Section 7.2. The encryption mode is enabled and the key and plaintext bytes are loaded into the key and the state memory of the AES crypto engine respectively. After the AES initialization is finished, the microcontroller signalizes this state which in turn allows the control computer to prepare the fault board for an attack. In this context, the clock glitch unit is armed, meaning the fault board is sensitive to its trigger input. The execution of the test application is resumed as soon as the fault board is ready.

- **Trigger signal.** To ensure a cycle-accurate synchronization between the fault board injecting the clock glitch and the microcontroller performing the AES encryption which should be hit by the attack, the microcontroller rises its trigger output signal. According to the attack settings of the fault board, the clock glitch is inserted after the predefined number of clock cycles after the trigger event.

- **AES execution.** Immediately after the trigger signal is set by the microcontroller, the test application starts the AES encryption engine by setting the appropriate bit of the AES control register. For a period of 375 clock cycles, `nop` instructions are executed on the microcontroller to avoid any unpredictable side effects until the encryption procedure completes. Simultaneously, the clock glitch is inserted to the clock signal by the fault board in compliance with the predefined attack settings. As a consequence, one of the 375 clock cycles used by the AES crypto engine for encryption is tampered.

- **Result communication.** After the encryption is complete, the test application accesses the state memory of the AES crypto engine containing the 128-bit ciphertext

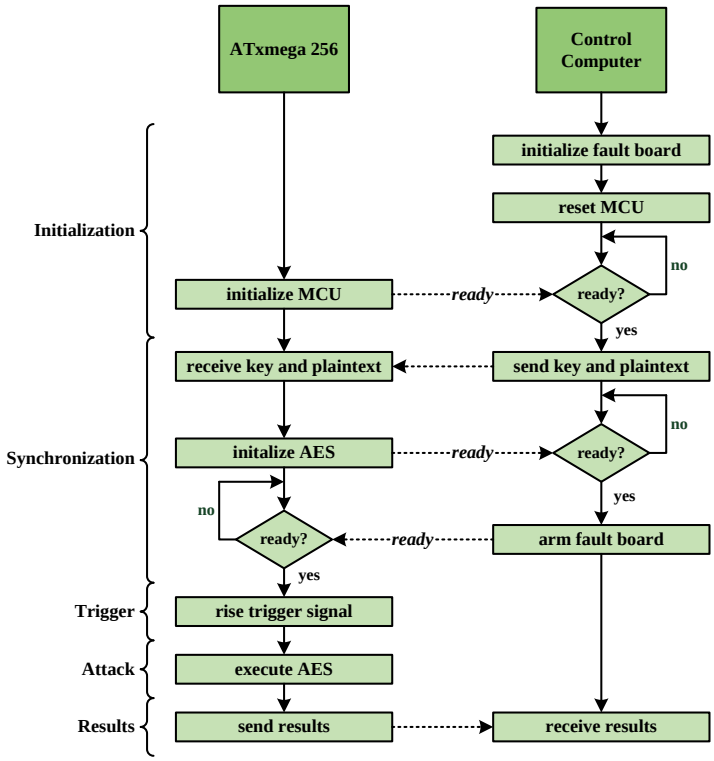

Figure 7.6: Sequence of operations performed during an attack of the AES crypto engine.

block and the key memory containing the 128-bit final round key. In this way, the values are transferred to the control computer using the UART interface. Based on a reference execution, without injecting a fault a differential fault analysis is possible. Analyzing the relation between the clock cylce tampered by the attack and the resulting faulty ciphertext in particular allows assumptions about the internal hardware architecture of the AES crypto engine.

Chapter 8

AES Crypto Engine: Attacks and Results

The following chapter gives a detailed description of the attacks performed on the AES crypto engine of the ATxmega 256 based on the attack scenario stated in Section 7.3. Furthermore, the obtained erroneous behavior of the AES crypto engine is analyzed by using faulty ciphertexts in order to draw meaningful conclusions about internal hardware structures. At the beginning, vulnerable hardware parts of the AES crypto engine are identified by investigating faulty ciphertexts on the basis of differential fault analysis. Consequently, a black-box characterization of the AES hardware implementation is possible, providing the necessary information for defining a fault model. In contrast to implicitly assuming a specific fault model for an attack, our approach starts with defining fault models based on the results, obtained in our practical experiments. The identified fault characteristics of the AES crypto engine result in two key retrieval attacks. The fault model for both attacks relies on a manipulation of the S-Box substitution during the final AES encryption round. By applying clock glitch attacks during the S-Box substitution, it is possible to change specific values within the AES state. In this context, we show how the entire AES key can be derived from 100 faulty ciphertexts with a probability of above 92 % percent. Additionally, we present how 14 faulty ciphertexts can be sufficient for retrieving the entire key with a probability of above 99 % provided that one of the 16 AES key bytes is known or correctly guessed.

8.1 Fault-Based Black-Box Characterization

Attacking a hardware implementation of a cryptographic algorithm without having detailed information about the hardware structure requires a more sophisticate approach compared to attacks of known implementations. In case of attacking the AES crypto engine of the ATxmega 256, the only information is given by the fact that the AES algorithm and its underlying operations are known. Thus, only rough assumptions about the encryption process are possible, knowing only the sequence of operations necessary for encryption or decryption and the number of cycles required by the AES crypt engine in order to encrypt or decrypt a 128-bit data block. Fault-based black-box characterization aims at retrieving hardware implementation details based on the error propagation of a fault induced during a single clock cycle. The relation between the affected clock cycle and corresponding erroneous ciphertext allows a more precise specification of AES hardware

implementation.

8.1.1 Attack Procedure

The attack procedure used in order to perform a block-box characterization of the AES crypto engine complies with the attack scenario presented in Section 7.3. The entire encryption procedure of a 128-bit data block requires 375 cycles. Depending on which clock cycle is manipulated by inserting a clock glitch, different active circuit parts of the AES crypto engine can be affected. The second important attack parameter is the glitch period T_{Glitch}. In case of inserting a clock glitch, T_{Glitch} defines the manipulated delay between two rising clock edges. The resulting erroneous behavior depends on the path delay of the combinational logic as well as on the processed values. Roughly speaking, the attacked clock cycle specifies in which part of the circuit a fault is injected. In this context, T_{Glitch} defines how the injected fault influences the circuit and consequently the processed data. To determine the influence of a single clock glitch inserted during AES encryption, a separate attack is performed for each of the 375 clock cycles. Moreover, the clock glitch period T_{Glitch} is varied between 5 ns and 12.5 ns. To simplify result evaluation and to avoid varying effects caused by different processed data, the same 128-bit plaintext block and the same 128-bit key are used for all attacks. The calculated ciphertexts and the last round key accessible in the key memory of the AES crypto engine are transferred to the control computer after each attack and are compared to a reference result in order to identify injected faults. For the following black-box characterization differences between the correct and a faulty ciphertext are analyzed on a byte level. This implies that a ciphertext byte is stated as faulty if at least one bit differs from the corresponding correct ciphertext byte.

8.1.2 Attack Results

To identify the glitch period T_{Glitch} for which the AES crypto engine produces the highest number of corrupted bytes within the ciphertexts, repeated attacks with a variation of the glitch period T_{Glitch} are performed. Results show that attacking the AES encryption procedure using a glitch period T_{Glitch} of 5.7 ns leads to a maximum number of induced faults. Figure 8.1 illustrates the faults in individual bytes of the resulting ciphertext after inducing a clock glitch in the corresponding clock cycle. For comparison, attacking clock cycle 320 influences byte 2, 5, 8, and 15 in the final ciphertext where a clock glitch in cycle 340 has no effects on the ciphertext. Based on this fault illustration, a characterization of the AES hardware implementation is possible as follows.

Key schedule. In order to identify faults in the ciphertext relying on manipulation of the key schedule, differential fault analysis of the final round key bytes is used. By comparing the final round key after an attack with the correct final round key, we are able to identify the time period during encryption when the key schedule is performed. Faults injected in the key schedule procedure reappear in the form of faults in the ciphertexts. Shown in Figure 8.1, the affected clock cycles are: 263 to 267 for the eighth round key, 299 to 301 for the ninth round key, and 333 to 335 for the final round key.

Substitute bytes. Beside the faults induced during the key schedule, the other faults shown in Figure 8.1 appear in similar intervals (clock cycle 244 to 259, 278 to 293, 312 to 327, and 345 to 360). In each of these intervals, fault injection is possible in 16 consecutive clock cycles leading to the assumption that the same AES operation is executed in these intervals. Therefore, each of these intervals belong to a specific encryption round.

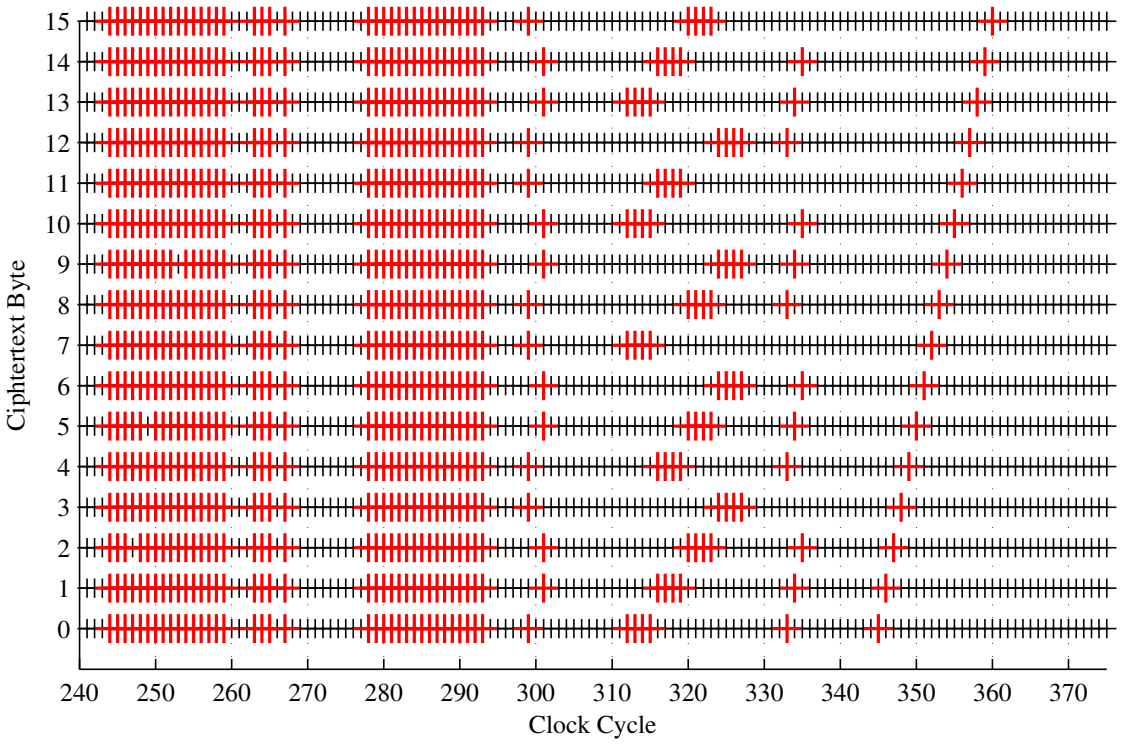

Figure 8.1: Faults in the ciphertext bytes after attacking each clock cycle of AES encryption. The attacked clock cycle is plotted on the x-axis and the resulting ciphertext bytes on the y-axis where the bold marker symbol + indicates a fault in the ciphertext.

Depending on the attacked round, different fault distribution within the ciphertext can be explained by the varying number of AES operations performed after the clock glitch attack. The last sequence between clock cycle 345 and 360 shows that only one byte is influenced in each clock cycle. If we now look at cycle 312, four ciphertext bytes are repetitively affected in the sequence of 16 clock cycles. By expecting that, again, only one byte is modified by the attack, the following considerations are necessary in order to define a relation between the fault distribution and the performed AES operation. First, each byte is influenced in four consecutive clock cycles. Second, four varying bytes are influenced by one clock glitch. Both observations can be explained by the fact that one *MixColumns* operation is performed after a single AES state byte was manipulated by the attack. Consequently, the faulty value of one byte is spread by *MixColumns* to four bytes of the state. Additionally, Figure 8.2 illustrates the pattern of the influenced bytes within the AES state between clock cycle 312 and 327. The distribution of faulty bytes within the state shows that the faults are injected before one *ShiftRows* operation. By combining all observations made about the AES encryption procedure, the *SubstituteBytes* exposed as the attacked operation, which is confirmed by the following assumptions and considerations:

- Only one S-Box is implemented in hardware, and is sequentially used to substitute one state byte in each clock cycle. Therefore, 16 clock cycles are required in each AES round to perform *SubstituteBytes*. Moreover, consecutive attacks between clock cycle 345 and 360 show that the AES bytes are consecutively manipulated.

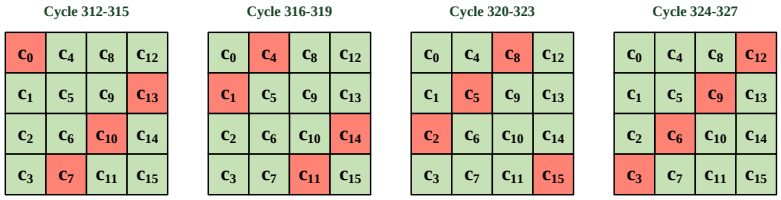

Figure 8.2: Distribution of faulty ciphertext bytes within the AES state, depending on the attacked clock cycle.

- The *MixColumns* operation is omitted in the final encryption round. Consequently, a single fault injected during one clock cycle of *SubstituteBytes* in the final round is not spread over an entire column of the state.
- Based on the resulting pattern of fault propagation between cycle 312 and 327, only one *MixColumns* operation and only one *ShiftRows* operation are subsequently performed to the attack. Regardless of which byte of the column is affected by the attack, *MixColumns* spreads the fault over the entire column.
- As no additional *MixColumns* operation is performed after *AddRoundKey* in the ninth round, *AddRoundKey* appeared to be not affected by the attacks.
- Only one *ShiftRows* operation is performed after the attack on *SubstituteBytes* in the ninth round in clock cycle 312 and 327. Based on the fault distribution depicted in Figure 8.1, this *ShiftRows* operation is executed before *SubstituteBytes* of the final round.
- Deviating from the sequence of AES operations given by the AES specification in Chapter 7, *ShiftRows* is executed before *SubstituteBytes* as the first operation in each round.

Beside the key schedule of the AES crypto engine, *SubstituteBytes* is the only AES operation influenced by clock glitch attacks. As *SubstituteBytes* is used within the key schedule, it is very likely that the resulting faults in the round key bytes are based on the same manipulation of the S-Box substitution. The black-box characterization of the AES crypto engine leads to the encryption sequence illustrated in Figure 8.3 for the ninth and the final AES round. The key schedule and the *ShiftRows* operation are performed first in each round followed by the vulnerable *SubstituteBytes* operation. *MixColumns* and *AddRoundKey* are not influenced by the performed clock glitch attacks. The precisely classified faults during *SubstituteBytes* and the direct relation between the attacked clock cycles and the corresponding modified byte provide the basis for the key retrieval attacks presented in the following section.

8.2 Key Retrieval Attack

This section describes the key retrieval attacks derived from the obtained erroneous behavior of the AES crypto engine based on the attacks performed during the black-box characterization in Section 8.1. For the black-box characterization, we analyze faults on byte level meaning that a single bit fault is sufficient to imply a ciphertext byte being faulty. In order to define a fault model for the key retrieval attacks, we focus on a refined

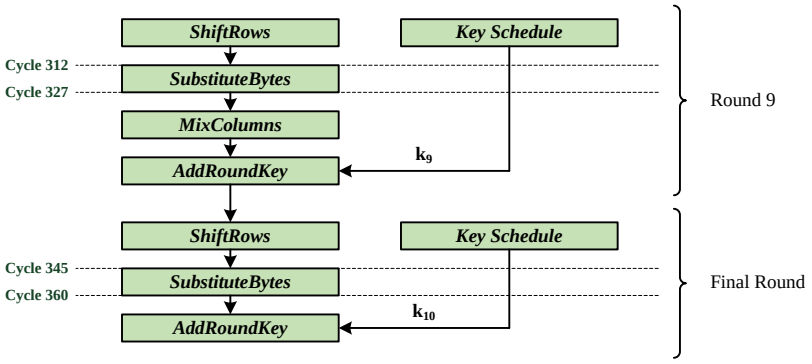

Figure 8.3: Fault-based characterization of the encryption sequence for the ninth and the final AES round.

analysis of faults injected during the *SubstituteBytes* operation of the final AES encryption round. The injected faults are therefore analyzed on the bit level as a function of the glitch period T_{Glitch}. Depending on T_{Glitch} of the inserted clock glitch during *SubstituteBytes*, different erroneous S-box output can be generated. As a result, wrong byte values are inserted by *SubstituteBytes* into the AES state matrix before the *AddRoundKey* operation adds the final round key to the state. In the following, we present the obtainable output values of the S-box and how these values can be used to define the fault models for the key retrieval attacks. Moreover, a detailed description of the attacks and the results of our practical experiments are given.

8.2.1 Bitwise Fault Analysis of the S-Box Output

As shown by the fault distribution in Figure 8.2, a single ciphertext byte can be manipulated by attacking the *SubstituteBytes* operation of the final AES round between clock cycle 345 to 360. Thus, inserting a glitch in one of the clock cycles between 345 to 360 leads to a fault in the corresponding state byte a_0 to a_{15}. After a fault is injected into one of the state bytes, the only operation which changes the value of this byte subsequently to *SubstituteBytes* is the final *AddRoundKey* operation. In the applied attack scenario, the only way of fault identification during the encryption procedure of the AES crypto engine of the ATxmega 256 is given through the resulting erroneous ciphertext. In order to determine the induced faults straight after *SubstituteBytes*, the final round key is again xored to the resulting ciphertext. As the XOR operation is its own inverse, the resulting values are equivalent to the values of the state before *AddRoundKey* which in turn represent the output values of the S-box after *SubstituteBytes*. Consequently, the erroneous S-box output is accessible if a known key is used for the fault analysis.

By applying clock glitch attacks to the *SubstituteBytes* operation, timing violations of the combinational path of the S-box logic lead to wrong output values. Considering a single clock glitch attack during the S-box operation of one state byte, the resulting erroneous value depends on a relation between the glitch period T_{Glitch} and the transition of the previous S-box output to the new output value for the actual byte. The glitch period T_{Glitch} defines the point in time when the S-box calculation is hit by the attack and furthermore, at which state of the S-box output transition the faulty value is written to the AES state.

To analyze the influence of the glitch period T_{Glitch} on the resulting faulty S-box output,

Table 8.1: S-Box output values for state byte a_4 after inserting a single clock glitch during *SubstituteBytes* with different values for T_{Glitch}.

$T_{Glitch,min}$	ΔT_{Glitch}	S-Box Output
ns	ns	
4.50	0.62	10101010
5.16	1.16	01010101
6.36	0.44	01010100
6.84	0.12	01000100
7.00	0.08	00000100
7.12	0.64	00000000
7.80	0.08	00100000
7.92	0.44	00101000
8.52	0.32	00101010
8.88	3.62	10101010

the attack procedure is based on the scenario given in Section 7.3. A single clock glitch is inserted during the *SubstituteBytes* operation of state byte a_4 in clock cycle 349 of the AES encryption. To define the S-box output of the previous byte a_3 and the actual byte a_4 for the *SubstituteBytes* operation of the final round, a precalculated key and ciphertext pair is used. Therefore, the key and ciphertext are chosen in a way that the S-box output in the final encryption round for a_3 is set to 0b01010101 and for a_4 to 0b10101010 in binary notation. Using these values ensures that each bit has to be toggled during the S-box output calculation for a_4. By varying the glitch period T_{Glitch} between 4.5 ns and 12.5 ns, a set of faulty ciphertexts is generated. To determine the faulty value of the S-box output, the key byte k_4 of the final round key is xored to corresponding byte a_4 of the ciphertext for each of these ciphertexts. Table 8.1 shows the resulting erroneous S-box outputs for a_4 as a function of T_{Glitch}. The time interval for a specific output value of the S-box is given by ΔT_{Glitch} starting from $T_{Glitch,min}$. Explained by the fact that the S-box calculation is not started until T_{Glitch} reaches 5.16 ns, the correct output for the AES state byte a_4 of 0b10101010 is generated. The S-box calculation finishes after 8.88 ns where in turn an inserted clock glitch has no effect on the output. Between 5.16 ns and 8.88 ns one can observe a transition of the individual bits from the previous S-box output of a_3 (0b01010101) to the correct output for a_4 (0b10101010). In this range of T_{Glitch} two particularly interesting effects occur. First, starting at 5.16 ns, the previous value of the S-box from a_3 is again inserted to the state byte a_4, leading to a duplicated entry in the AES state after the *SubstituteBytes* operation. Second, starting at 7.12 ns, all bits of the S-box output are set to zero, leading to a zero entry in the AES state after the *SubstituteBytes* operation. Both effects can be obtained in a sufficient time interval for T_{Glitch} of either 1.12 ns for the duplicate entry or 0.64 ns for the zero entry in case of the S-box output transition from 0b01010101 to 0b10101010.

In order to define generally applicable ranges for T_{Glitch}, we performed additional clock glitch attacks on the S-box output transition for the state byte a_4 at clock cycle 349. Precalculated key and ciphertext pairs are used to specify the S-box output for the state bytes a_3 and a_4 in the final encryption round. In doing so, all possible S-box output values for a_3 between 0 and 255 are used. Corresponding to these values, the output for

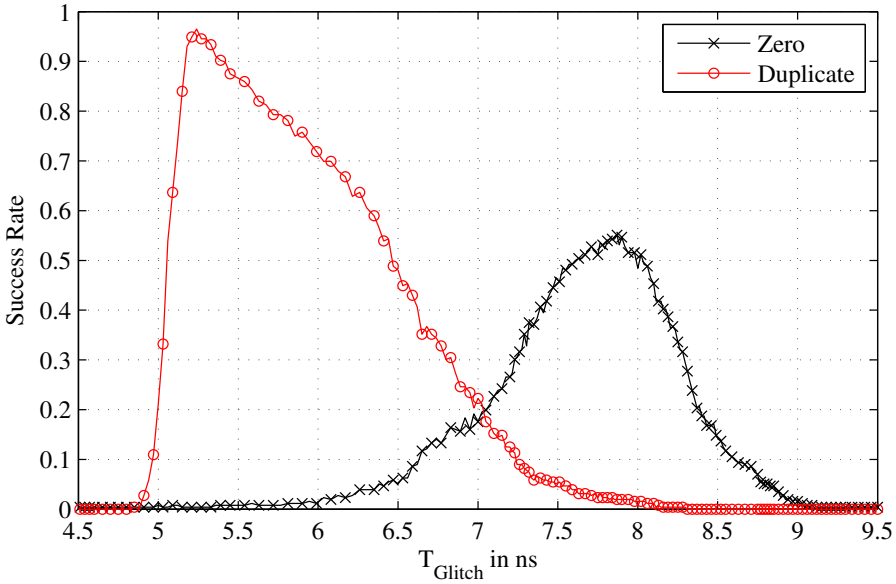

Figure 8.4: Success rate of the clock glitch attack on the *SubstituteBytes* operation of one AES state byte in the final encryption round. Successful attacks are separated by means of either setting the state byte to zero or to the previous S-box output value.

a_4 is chosen in a way, that the inverse value has to be calculated by the S-box operation. For example, if the S-box output for a_3 is 0b00000001, a S-box transition to 0b11111110 is required. For each of the resulting 256 key and ciphertext pairs, single clock glitch attacks with varying glitch period T_{Glitch} are performed in cycle 349 of the AES encryption procedure. By evaluating the time intervals of T_{Glitch} in which the value of the state byte a_4 is either set to the S-box output of a_3 or to zero, it is possible to identify values for the glitch period T_{Glitch} with high success rates. Based on these results, Figure 8.4 illustrates the success rates for changing the value of a_4 either to zero or to the duplicated output of a_3. Starting at a glitch period T_{Glitch} of about 5 ns, the quickly increasing probability of producing a duplicate S-box output shows that this effect occurs almost regardless of the previous or the approaching S-box output. Therefore, the highest success rate of 96 % can be achieved with a glitch period T_{Glitch} of 5.24 ns. With 55 % at a glitch period T_{Glitch} of 7.87 ns, the maximal success rate for setting the state entry to zero is considerably smaller than for getting a duplicate. This effect can be explained by a stronger variation of the time interval where the S-box output is set to zero. Depending on the previous and the approaching S-box value, varying timing behavior of the transition between these two values is given. In addition to the evaluation of the S-box output for transitions between inverse values where each bit of the S-box output has to be flipped, the same behavior can also be observed when the transitions between other values is attacked. Therefore, the time intervals for getting duplicate S-box output or setting the S-box output to zero, also exist for S-box transitions between two values where a_3 and a_4 are equal or at least one bit stays the same.

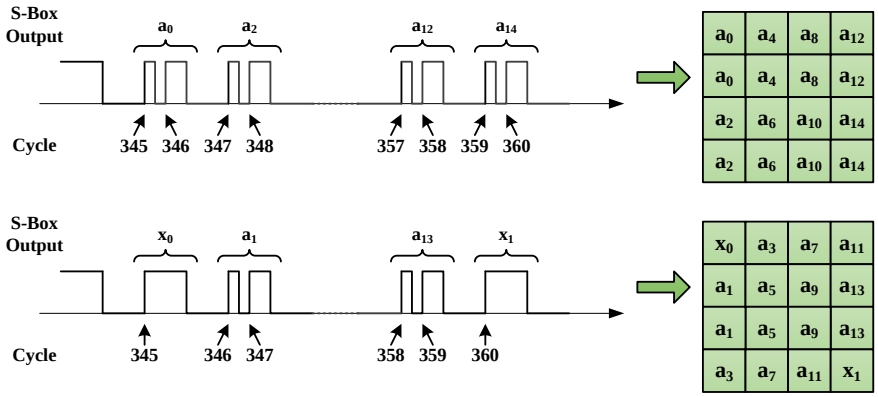

Figure 8.5: Consecutive inserted clock glitches starting at cycle 345, respectively at cycle 346 and the corresponding distribution of the duplicated values within the AES state matrix after the attacked *SubstituteBytes* operation.

8.2.2 Attack 1: Duplicating AES State Entries

The first attack we present is based on the observation of duplicate S-box output entries in the AES state after applying clock glitch attacks to the *SubstituteBytes* operation in the final encryption round. So far, only one clock glitch was inserted during an attack, leading to a single fault on one byte of the state. The *SubstituteBytes* operation of the final round requires 16 consecutive clock cycles and in each cycle one state byte is substituted. By inserting consecutive clock glitches during the S-box operation between clock cycle 345 and 360, duplicate entries can be produced within the entire AES state. The clock glitch period T_{Glitch} of 5.24 ns is therefore chosen based on the maximal success rate determined in previous attack experiments as shown in Figure 8.4. By applying eight clock glitches during the S-box operation starting at clock cycle 345, it is possible to produce consecutive duplicates within the AES state. The entries $a_0, \ldots, a_{15}$ describe the values within the AES state after *SubstituteBytes* of a reference execution without applying clock glitches. In case of using the same key and ciphertext pair and inserting eight consecutive clock glitches starting at cycle 345, the 16 bytes of the AES state after the attacked *SubstituteBytes* operation are organized as follows: $a_0, a_0, a_2, a_2, a_4, a_4, a_6, a_6, a_8, a_8, a_{10}, a_{10}, a_{12}, a_{12}, a_{14}, a_{14}$. As a result, only eight different values are present within the state after an attack with a distinct distribution of duplicate entries. In each case, two consecutive state bytes have the same value. Starting the attack with seven consecutive inserted clock glitches at clock cycle 347 leads to a similar behavior. The duplicates are shifted by one position within the state resulting in the following distribution of the values: $x_0, a_1, a_1, a_3, a_3, a_5, a_5, a_7, a_7, a_9, a_9, a_{11}, a_{11}, a_{13}, a_{13}, x_1$. The two bytes x_0 and x_1 represent variable values without a relation to other values of the state after an attack. Figure 8.5 illustrates the correlation between the sequence of inserted clock glitches and the corresponding distribution of duplicate values in the state. In case of inserting a clock glitch, the combinational logic of the S-box implementation is in a settled condition at the first rising edge of the clock signal. The S-box operation for the corresponding state byte produces a correct result. At the next rising edge of the clock signal, a violation of the combinational path leads to a faulty S-box output. With the appropriate choice for the delay between these tow rising edges specified by the glitch period T_{Glitch}, it is possible to achieve the desired S-box output values.

So far, knowledge of the encryption key is necessary for analyzing the behavior of the AES crypto engine to fault attacks. Information about the faulty S-box outputs of *SubstituteBytes* in the final encryption round can be retrieved by xoring the final round key to the corresponding faulty ciphtertext. However, the key is usually not known to an adversary who tries to break a cryptographic application. For a realistic and practical attack scenario we define the following constraints and preconditions:

- An adversary has access to the device and is able to induce faults during the *SubstituteBytes* operation in the final round, in order to produce the aforementioned duplicate values in the AES state before the final round key is added.
- The only accessible information of the encryption procedure is the resulting ciphertext of the encryption of an unknown key and plaintext pair. Therefore, key and plaintext remain hidden from an adversary.
- The same key is used during the encryption of several 128-bit plaintext blocks. Plaintext data blocks can vary in each attacked encryption procedure.
- As shown in Figure 8.5, at least two independent attacks of the *SubstituteBytes* operation are necessary: first, starting at clock cycle 346 with eight consecutive inserted clock glitches and second, starting at clock cycle 347 with seven consecutive inserted clock glitches.

The duplicate values in the AES state before adding the final round key result in a direct relation between two consecutive ciphertext bytes. Consequently, it is possible to calculate the final round key bytes. For the description of this relation and the resulting method of calculating the final round key, the following terms are used:

- The final round key bytes are defined by $k_0, \ldots, k_{15}$.
- The erroneous ciphtertext bytes, corresponding to the duplicate state entries $a_0, a_2, a_4, a_6, a_8, a_{10}, a_{12}, a_{14}$ after attacking the *SubstituteBytes* operation are defined by $c_{a,0}, \ldots, c_{a,15}$.
- The erroneous ciphtertext bytes, corresponding to the duplicate state entries $a_1, a_3, a_5, a_7, a_9, a_{11}, a_{13}$ and x_0, x_1 after attacking the *SubstituteBytes* operation are defined by $c_{b,0}, \ldots, c_{b,15}$.

Two consecutive bytes of the ciphertext $c_{a,0}, \ldots, c_{a,15}$ build a relation, based on duplicated entries in the state after *SubstituteBytes*. This relation can be described by Equation 8.1. The same relation is valid for the ciphtertext bytes $c_{b,0}, \ldots, c_{b,15}$. In this context, Equation 8.2 is used by considering the offset of the duplicated entries in the state after *SubstituteBytes*. On condition that both ciphertexts were calculated using the the same key, Equation 8.3 can be used to retrieve the final round key bytes. By applying Equation 8.3 to the two faulty ciphtertexts, each key byte $k_1, \ldots, k_{15}$ can be derived from the value of k_0. This implies that one key byte has to be known, in order to calculate the remaining 15 bytes of the final round key. Either 256 tries are necessary or a known plaintext/ciphertext pair can be used for determining the missing key byte. Additionally, the second attack on the AES crypto engine of the ATxmega 256, presented in Section 8.2.3, can be applied for retrieving the missing key byte. Once the final round key is

revealed, the inverse AES key schedule can be used subsequently for calculating the initial AES key.

$$k_i \oplus c_{a,i} = k_{i+1} \oplus c_{a,i+1} \qquad (i = 0, 2, \ldots, 12, 14) \tag{8.1}$$

$$k_i \oplus c_{b,i} = k_{i+1} \oplus c_{b,i+1} \qquad (i = 1, 3, \ldots, 11, 13) \tag{8.2}$$

$$\begin{aligned} k_i &= c_{a,i-1} \oplus c_{a,i} \oplus k_{i-1} \qquad (i = 1, 3, \ldots, 13, 15) \\ k_i &= c_{b,i-1} \oplus c_{b,i} \oplus k_{i-1} \qquad (i = 2, 4, \ldots, 12, 14) \end{aligned} \tag{8.3}$$

The capability of this attack for successfully calculating the correct key bytes relies on the assumption that each inserted clock glitch during the *SubstituteBytes* operation induces the desired fault behavior for producing duplicate entries in the AES state. As depicted in Figure 8.4, a success rate of 96 % can be achieved for a single byte when generating one duplicate entry in the state. To verify the success rate for retrieving the entire key, we performed 2000 attacks on the encryption procedure. The glitch period T_{Glitch} of 5.24 ns is used for the consecutive inserted clock glitches. By starting the attack at clock cycle 345 with 8 glitches, 1000 erroneous ciphertexts (c_a) are generated. The other half of erroneous ciphertexts (c_b) is generated by starting the attack at clock cycle 346 with 7 glitches. In each attack, the same key is used to encrypt randomly chosen 128-bit plaintext blocks. By applying Equation 8.3 to the resulting erroneous ciphertexts, 1000 possible keys are calculated. For each calculation a pair of c_a and c_b is used and the first key byte k_0 is assumed to be known. By comparing each calculated key with the correct key, it is possible to specify a success rate for retrieving the correct key. Table 8.2 shows the individual rates for successfully retrieving the key bytes $k_0, \ldots, k_{15}$. Due to the fact that k_0 is assumed to be known, the success rate for k_0 is set to 100 %. Table 8.2 additionally presents the success rates when using either three, five, seven, nine, or eleven pairs of ciphertexts c_a and c_b. Corresponding to the number of used pairs, a majority decision can be used for individually selecting each key byte. Equation 8.3 shows that the result for a key byte k_i depends on the value of the previous calculated key byte k_{i-1}. In this context, an error propagation is given, which leads to a continuously decreasing success rate between the first and the last calculated key byte. When using more then one c_a/c_b pair, it is possible to select the most probable value for each key byte before its value is used for the calculation of the next key byte. In doing so, a success rate higher than 99 % for determining the entire key can be achieved by only using seven c_a/c_b pairs. However, it is assumed that the first key byte k_0 is known.

8.2.3 Attack 2: Zeroing AES State Entries

In addition to the key retrieval attack, based on duplicated AES state entries 8.2.2, another approach for an attack is given by the fact, that with a proper choice for the glitch period T_{Glitch}, the S-box output can be forced to zero. As shown by the bitwise fault analysis of the S-box output in Section 8.2.1, a single clock glitch during the *SubstituteBytes* operation in the final encryption round can lead to the effect of setting the appropriate state byte to zero before *AddRoundKey* adds the final round key bytes to the state. If that occurs, the key byte is directly available in the resulting ciphertext regardless of all previous AES transformations. By inserting consecutive clock glitches between cycle 344 and 359 during the S-box operation, it is possible to achieve the desired behavior for several AES

Table 8.2: Success rates for retrieving the correct final round key bytes, when using the given number of erroneous ciphertext pairs, c_a and c_b. Key byte k_0 is assumed to be known.

Key Byte	Success Rate					
	1 c_a/c_b	3 c_a/c_b	5 c_a/c_b	7 c_a/c_b	9 c_a/c_b	11 c_a/c_b
k_0	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %
k_1	98.2 %	99.7 %	100.0 %	100.0 %	100.0 %	100.0 %
k_2	93.8 %	99.1 %	99.8 %	100.0 %	100.0 %	100.0 %
k_3	91.1 %	99.0 %	99.7 %	100.0 %	100.0 %	100.0 %
k_4	81.9 %	96.8 %	99.3 %	100.0 %	100.0 %	100.0 %
k_5	78.9 %	96.5 %	99.1 %	100.0 %	100.0 %	100.0 %
k_6	76.5 %	96.2 %	99.1 %	100.0 %	100.0 %	100.0 %
k_7	75.3 %	96.1 %	99.1 %	100.0 %	100.0 %	100.0 %
k_8	71.9 %	95.5 %	99.1 %	99.9 %	100.0 %	100.0 %
k_9	62.5 %	94.1 %	98.9 %	99.9 %	100.0 %	100.0 %
k_{10}	56.1 %	91.4 %	98.6 %	99.9 %	100.0 %	100.0 %
k_{11}	49.1 %	88.2 %	98.2 %	99.9 %	99.9 %	100.0 %
k_{12}	43.6 %	85.9 %	97.8 %	99.9 %	99.9 %	100.0 %
k_{13}	38.2 %	85.3 %	97.6 %	99.8 %	99.9 %	100.0 %
k_{14}	35.0 %	84.3 %	97.3 %	99.8 %	99.8 %	100.0 %
k_{15}	30.1 %	79.9 %	96.6 %	99.7 %	99.8 %	100.0 %

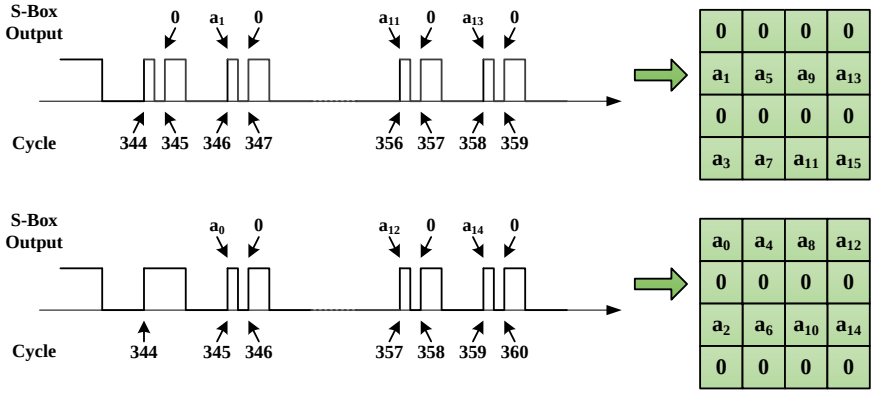

Figure 8.6: Consecutive inserted clock glitches starting at cycle 344, respectively at cycle 345 and the corresponding distribution of the zeros within the AES state matrix after the attacked *SubstituteBytes* operation.

state bytes by a single attack run. The clock glitch period T_{Glitch} of 7.12 ns is chosen according to the maximal success rate determined during the bitwise fault analysis of the S-box operation illustrated in Figure 8.4. Under the assumption that $a_0, \ldots, a_{15}$ describe the values within the AES state after *SubstituteBytes* of a reference execution without applying clock glitches. When eight clock glitches are inserted consecutively starting at cycle 344, the values $a_0, a_2, a_4, a_6, a_8, a_{10}, a_{12}, a_{14}$ are zero. Starting the attack one clock cycle later, at cycle 345, the values $a_1, a_3, a_5, a_7, a_9, a_{11}, a_{13}, a_{15}$ can be forced to zero. Two faulty ciphertexts c_a and c_b are necessary for retrieving the final round key. Figure 8.6 illustrates the correlation between the sequence of inserted clock glitches and the values of the state which are set to zero after the attacked *SubstituteBytes* operation of the final encryption round. Depending on the glitch period T_{Glitch}, every second rising edge of the clock signal violates the timing constraints of the combinational path of the S-box operation, further leading to the desired S-box output of zero. In order to define a practicable attack procedure based on the resulting erroneous behavior of *SubstituteBytes*, the following constraints and preconditions are given:

- An adversary has access to the device and is able to induce faults during the *SubstituteBytes* operation in the final round in order to produce the aforementioned zeros in the AES state before the final round key is added.

- The only accessible information of the encryption procedure is the resulting ciphertext of the encryption of an unknown key and plaintext pair. Therefore, key and plaintext remain hidden from an adversary.

- The same key is used during the encryption of several 128-bit plaintext blocks. Plaintext data blocks can vary in each attacked encryption procedure.

- As shown in Figure 8.6, at least two independent attacks of the *SubstituteBytes* operation are necessary, starting at clock cycle 344 and at 345 respectively with eight consecutive inserted clock glitches.

Assuming that every inserted clock glitch leads to the desired behavior of the S-box operation, two faulty ciphertexts c_a and c_b are sufficient in order to retrieve the final round

Table 8.3: Success rates for retrieving the correct final round key bytes when using the given number of erroneous ciphertext pairs, c_a and c_b.

Key Byte	Success Rate						
	1 c_a/c_b	5 c_a/c_b	10 c_a/c_b	20 c_a/c_b	30 c_a/c_b	40 c_a/c_b	50 c_a/c_b
k_0	55.0 %	79.9 %	94.1 %	98.8 %	99.8 %	99.9 %	99.9 %
k_1	25.8 %	37.6 %	62.9 %	77.4 %	89.5 %	93.7 %	95.0 %
k_2	26.7 %	68.4 %	78.7 %	91.9 %	96.0 %	98.3 %	98.9 %
k_3	26.0 %	36.0 %	58.5 %	76.6 %	82.8 %	87.4 %	92.9 %
k_4	49.0 %	71.2 %	89.5 %	97.0 %	99.6 %	99.7 %	99.9 %
k_5	24.1 %	54.5 %	71.0 %	82.8 %	89.3 %	91.7 %	94.2 %
k_6	19.8 %	24.2 %	49.0 %	69.4 %	79.3 %	87.6 %	93.3 %
k_7	26.7 %	39.4 %	60.7 %	81.0 %	86.0 %	90.9 %	94.3 %
k_8	45.5 %	82.8 %	92.5 %	96.5 %	99.0 %	99.6 %	99.4 %
k_9	22.7 %	32.0 %	59.4 %	80.3 %	91.6 %	96.7 %	99.9 %
k_{10}	37.0 %	59.7 %	82.9 %	96.7 %	98.9 %	99.7 %	99.7 %
k_{11}	20.4 %	27.5 %	57.5 %	76.9 %	90.2 %	93.9 %	97.1 %
k_{12}	45.1 %	70.2 %	88.8 %	96.6 %	99.8 %	99.8 %	99.9 %
k_{13}	28.3 %	40.8 %	72.0 %	88.1 %	95.5 %	97.7 %	99.3 %
k_{14}	24.1 %	33.8 %	64.7 %	83.3 %	93.7 %	96.8 %	98.4 %
k_{15}	20.9 %	30.7 %	54.9 %	77.7 %	88.6 %	92.9 %	95.1 %

key. In this case, c_a expose the final round key bytes $k_0, k_2, k_4, k_6, k_8, k_{10}, k_{12}, k_{14}$ and c_b exposes $k_1, k_3, k_5, k_7, k_9, k_{11}, k_{13}, k_{15}$. However, a stronger variation of the time interval where the S-box output is set to zero leads to a lower success rate compared to the first attack which is based on producing duplicate entries in the AES state. Therefore, the success rate for retrieving the entire key highly depends on the capability of the attack for setting each S-box output for the 16 S-box operations in the final encryption round to zero. To verify the achievable success rate for retrieving the entire key, we performed 2000 attacks on the encryption procedure, using a glitch period T_{Glitch} of 7.12 ns. The first 1000 faulty ciphertexts (c_a) are generated by inserting 8 consecutive clock glitches starting at clock cycle 344 and the second 1000 faulty ciphertexts (c_b) starting at clock cycle 345. The first column of Table 8.3 illustrates the average rates for successfully retrieving the individual key bytes $k_0, \ldots, k_{15}$, based on the 2000 faulty ciphertexts. It appears that the average success rates for the key bytes k_0, k_4, k_8, k_{12} are about twice as high compared to the success rates for the other key bytes. Assuming that an adversary is able to generate several faulty c_a/c_b ciphertext pairs, the success rates can be increased by using a majority decision. Table 8.3 additionally shows the corresponding success rates when using either 5, 10, 20, 30, 40, or 50 ciphtertext pairs to determine the most probable key bytes. The ciphertext pairs are randomly chosen from the test set of 2000 faulty ciphertexts and each evaluation is repeated 1000 times in order to get representative results. Results show that it is possible to retrieve the entire key with a probability higher than 92 %, using 50 faulty c_a/c_b ciphertext pairs.

8.2.4 Discussion and Comparison

The bitwise fault analysis of the S-box output when attacking the *SubstituteBytes* operation in the final encryption round allows to define two fault models for a key retrieval attack. Depending on the glitch period T_{Glitch}, it is either possible to produce duplicated values in the AES state by *SubstituteBytes* or AES state entries are set to zero before the final round key is added. With the approach of inducing duplicated entries, a success rate higher than 99 % for retrieving the correct key is achievable with only 14 faulty ciphertexts. Table 8.2 gives an overview about the capabilities of this attack. The only limitation is given by the fact that the first byte of the final round key k_0 has to be known in order to calculate all other key bytes. However, the second presented attack can be utilized to determine the first key byte. Results show that the rate for successfully retrieving k_0 is higher than 99 % when 60 additional faulty ciphertexts are used. This attack can be applied on the 16 consecutive executed S-box operations in order to reveal the entire AES key. In this case, 100 faulty ciphertexts lead to a success rate higher than 92 %. Table 8.3 presents the corresponding results. The difference in the number of necessary faulty ciphertexts and the lower success rate compared to the first attack is given by a higher variation of the time interval in which the S-box output is set to zero. These time intervals differ depending on the transition between the previous and the new output value of the S-box. Therefore, it is more challenging to define a general value for the glitch period T_{Glitch}. With both presented attacks, the final round key can be revealed and further used to calculate the AES key by applying the inverse AES key schedule.

Chapter 9

Conclusions

In this thesis we have analyzed the effects of non-invasive fault attacks on two different microcontroller platforms: ARM Cortex-M0 and Atmel ATxmega 256. Fault attacks based on clock glitch insertion and supply voltage manipulation were applied. In a variety of practical experiments we have identified and characterized common and individual faults for both MCUs by investigating the processing sequence of assembly instructions for the two-stage pipeline of the ATxmega 256 on the one hand, and for the three-stage pipeline of the Cortex-M0 on the other hand. The resulting behavior to clock glitch attacks on the instruction-fetch operation in particular was very reliable for both MCUs. Similar and well-definable attack parameters, almost regardless of the analyzed instruction could be revealed. In this context, we have shown that the fetch operation can be manipulated to the effect that the previously fetched instruction remains in the instruction fetch buffer instead of loading the new instruction. Under consideration that the attacked instruction is replaced by the previous one rather than entirely skipped, the observed faults in the fetch stage allow precise and deterministic modifications of the program flow. Consequently, an adversary is able to essentially influence implementations of cryptographic algorithms, e.g. by preventing verification routines from being executed or by skipping entire code segments.

Furthermore, we have observed that faults can be injected into the execution of instructions. In case of attacking the execution stage, data flow manipulation is possible and the resulting behavior highly depends on the attacked instruction. The results of the practical experiments showed that both microcontrollers can be forced to produce wrong calculation results when attacking the execution stage of the instruction pipeline. Particularly, faults in the execution of memory operations, e.g. load and store instructions, led to constant values either loaded from or written to memory. When using a single clock glitch during an attack, applicable fault models have to consider parallel processing of instructions depending on the individual pipeline structure of the MCU. Regardless of the investigated instruction and the attacked stage of processing, we were able to precisely define time intervals for the clock glitch period resulting in reliable and constant fault injection for all investigated scenarios.

A, to the author's knowledge novel approach, consisting of the combination of short-time underpowering and clock glitch insertion has been used to significantly improve the sensitivity to fault attacks on the investigated device. The applied supply voltage reduction led to an increased path delay of the combinational logic which, in turn, increased the sensitivity of the attacked device to fault injection, using clock glitches. In this context, we have identified a relation between the reduced voltage level and the time interval for

the clock glitch period leading to successful fault injections.

In the second part of this thesis we have analyzed the effects of clock glitch attacks on the AES crypto engine, a hardware accelerator for AES encryption/decryption, implemented on Atmel's ATxmega 256. Implementation-specific details and affected AES operations were identified during a fault-based black-box characterization of the AES crypto engine. By analyzing the resulting faults within the ciphertexts after consecutively inserting a single clock glitch in every execution cycle of the encryption procedure, the *SubstituteBytes* operation appeared to be reliably susceptible to fault injection using clock glitch attacks. Thus, we have observed that the *SubstituteBytes* operation can be forced either to generate duplicated values in the AES state or to set specific bytes of the AES state to zero depending on the clock glitch period. Based on these results, we have introduced two key-retrieval attacks targeting the final AES encryption round. Both attacks are very powerful due to the fact that only faulty ciphertexts are required to reveal the secret key.

Within the scope of investigating the potential of non-invasive fault attacks, we have identified several reliable fault models and vulnerabilities for both microcontrollers which make them potential targets for these kind of attacks. Underlined by the fact that relatively cheap equipment is sufficient for successfully performing clock glitch insertion and power supply manipulation, protection against fault attacks is necessarily required for security-relevant applications where microcontrollers are involved. The described attacks were implemented on unprotected devices and without any countermeasures against fault injection in order to identify common fault mechanisms and to understand potential threats rather than verify specific defense mechanisms. The aim of this work is to provide results which serve as a basis for implementing adequate countermeasures. The obtained vulnerabilities of the microcontrollers can already be considered during early development stages of secure software or hardware designs. However, countermeasures often come along with tradeoffs between implementation effort, performance losses and the level of protection. Therefore, proper identification of potential threats and a precise understanding of underlying fault injection mechanisms is necessary to provide efficient protection against fault attacks and to minimize the required implementation overhead.

Appendix A

Definitions

Abbreviations

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ASIC	Application-specific Integrated Circuit
BOD	Brown-Out Detection
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
DCM	Digital Clock Manager
DFA	Differential Fault Analysis
DIP	Dual Inline Package
DMA	Direct Memory Access
DPA	Differential Power Analysis
DUT	Device Under Test
EEPROM	Electrically Erasable Programmable Read-Only Memory
FPGA	Field-programmable Gate Array
MCU	Microcontroller Unit
PDI	Program and Debug Interface
PLL	Phase-Locked Loop
RFID	Radio-Frequency Identification
RISC	Reduced Instruction Set Computer
RSA	Rivest Shamir Adleman
SWD	Serial Wire Debug
SRAM	Static Random-Access Memory
TQFP	Thin Quad Flat Package
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
YAG	Yttrium Aluminum Garnet

Bibliography

[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] ARM Limited. Cortex-M0 Devices, Generic User Guide. Available online at `http://infocenter.arm.com/help/topic/com.arm.doc.dui0497a/DUI0497A_cortex_m0_r0p0_generic_ug.pdf`, 2009.

[3] Atmel Corporation. AVR1318: Using the XMEGA built-in AES accelerator. Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc8106.pdf`, April 2008.

[4] Atmel Corporation. 8-bit AVR Instruction Set. Available online at `http://www.atmel.com/images/doc0856.pdf`, 2010.

[5] Atmel Corporation. 8-bit Atmel XMEGA A Microcontroller, XMEGA A MANUAL. Available online at `http://www.atmel.com/images/doc8077.pdf`, 2012.

[6] 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.

[7] 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*, pages 105–114. IEEE, 2011.

[8] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[9] E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Advances in Cryptology CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer Berlin Heidelberg, 1997.

[10] H. Choukri and M. Tunstall. Round Reduction Using Faults. In L. Breveglieri and I. Koren, editors, *Second Workshop on Fault Diagnosis and Tolerance in Cryptography - FDTC 2005, Edinburgh, Scotland, UK, September 2, 2005, Proceedings*, September 2005.

[11] J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[12] A. Dehbaoui, A.-P. Mirbaha, N. Moro, J.-M. Dutertre, and A. Tria. Electromagnetic Glitch on the AES Round Counter. In *Constructive Side-Channel Analysis and Secure Design*, volume 7864 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin Heidelberg, 2013.

[13] A. Djellid-Ouar, G. Cathebras, and F. Bancel. Supply Voltage Glitches Effects on CMOS Circuits. In *Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006. International Conference on*, pages 257–261. IEEE, 2006.

[14] S. Endo, T. Sugawara, N. Homma, T. Aoki, and A. Satoh. An on-chip glitchy-clock generator and its applicataion to sage-error attack. In *Second International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE 2011), 24-25 February 2011, Darmstadt, Germany*, Workshop Proceedings COSADE 2011, pages 175–182, 2011.

[15] T. Fuhr, E. Jaulmes, V. Lomné, and A. Thillard. Fault Attacks on AES with Faulty Ciphertexts Only. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 108–118. IEEE, 2013.

[16] T. Fukunaga and J. Takahashi. Practical Fault Attack on a Cryptographic LSI with ISO/IEC 18033-3 Block Ciphers. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on*, pages 84–92. IEEE, 2009.

[17] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In *Cryptographic Hardware and Embedded Systems - CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer Berlin Heidelberg, 2001.

[18] M. Hutter and J.-M. Schmidt. The Temperature Side-Channel and Heating Fault Attacks. In *Smart Card Research and Advanced Applications - CARDIS 2013, 12th International Conference, Berlin, Germany, November 27-29, 2013, Proceedings.*, Lecture Notes in Computer Science, 2013.

[19] D. Karaklajic, J.-M. Schmidt, and I. Verbauwhede. Hardware Designer's Guide to Fault Attacks. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, pages 2295–2306, 2013.

[20] 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*, volume 4462 of *Lecture Notes in Computer Science*, pages 215–228. Springer Berlin Heidelberg, 2007.

[21] I. Kizhvatov. Side Channel Analysis of AVR XMEGA Crypto Engine. In *Proceedings of the 4th Workshop on Embedded Systems Security*, WESS '09, pages 8:1–8:7, New York, USA, 2009. ACM.

[22] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology CRYPTO 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer Berlin Heidelberg, 1999.

[23] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology CRYPTO 96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer Berlin Heidelberg, 1996.

[24] 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*, WOST'99, pages 9–20. USENIX Association, 1999.

[25] T. Korak. Investigation of Parameters Influencing the Success of Optical Fault Attacks. In *Foundations and Practice of Security*, Lecture Notes in Computer Science, pages 140–157. Springer International Publishing, 2014.

[26] Y. Li, Y.-i. Hayashi, A. Matsubara, N. Homma, T. Aoki, K. Ohta, and K. Sakiyama. Yet Another Fault-Based Leakage in Non-uniform Faulty Ciphertexts. In *Foundations and Practice of Security*, Lecture Notes in Computer Science, pages 272–287. Springer International Publishing, 2014.

[27] J. S. Melinger, S. Buchner, D. McMorrow, W. J. Stapor, T. R. Weatherford, A. Campbell, and H. Eisen. Critical Evaluation of the Pulsed Laser Method for Single Event Effects Testing and Fundamental Studies. *Nuclear Science, IEEE Transactions on*, pages 2574–2584, 1994.

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

[29] National Institute of Standards and Technology (NIST). Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001.

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

[31] NXP. UM10398, LPC111x/LPC11Cxx User manual. Available online at `http://www.nxp.com/documents/user_manual/UM10398.pdf`, 2014.

[32] C. Paar and J. Pelzl. The Advanced Encryption Standard (AES). In *Understanding Cryptography*, pages 87–121. Springer Berlin Heidelberg, 2010.

[33] J. Schmidt and C. Herbst. A Practical Fault Attack on Square and Multiply. In *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC '08. 5th Workshop on*, pages 53–58. IEEE, Aug 2008.

[34] J.-M. Schmidt and M. Hutter. Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results. In *Austrochip 2007, 15th Austrian Workhop on Microelectronics, 11 October 2007, Graz, Austria, Proceedings*, pages 61 – 67. Verlag der Technischen Universität Graz, 2007.

[35] 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.

[36] 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/`.

[37] S. P. Skorobogatov and R. J. Anderson. Optical Fault Induction Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer Berlin Heidelberg, 2003.

[38] K. T. Tan, S. H. Tan, and S. H. Ong. Functional Failure Analysis on Analog Device by Optical Beam Induced Current Technique. In *Physical amp; Failure Analysis of Integrated Circuits, 1997., Proceedings of the 1997 6th International Symposium on*, pages 296–301. IEEE, 1997.

[39] E. Trichina and R. Korkikyan. Multi Fault Laser Attacks on Protected CRT-RSA. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pages 75–86. IEEE, 2010.

[40] Xilinx. Spartan-6 FPGA Clocking Resources User Guide. Available online at `http://www.xilinx.com/support/documentation/user_guides/ug382.pdf`, 2013.

[41] S.-M. Yen and M. Joye. Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis. *Computers, IEEE Transactions on*, pages 967–970, 2000.

[42] J. Yiu. *The Definitive Guide to the ARM Cortex-M0, 1st Edition.* Academic Press (Elsevier), 2011.

[43] 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.