FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Active non-invasive attack on a microcontroller by fault injection
Student:	Jakub Kučera
Supervisor:	Ing. Jiří Buček, Ph.D.
Study program:	Informatics
Branch / specialization:	Information Security 2021
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

Active non-invasive attacks are a type of hardware attack that aims to disturb the normal running of the target so that it performs a faulty operation. The resulting error in control flow or data can be exploited to circumvent security measures, for example to reveal a secret key of a cipher.

* Study non-invasive fault injection attacks on microcontrollers, focus on voltage glitching.
* Study existing voltage glitching tools such as NewAE ChipWhisperer Nano.
* Design and implement a voltage glitching attack using a FPGA to control the glitch parameters and a simple crowbar circuit for glitch insertion. Optionally, also implement clock glitching.
* Use the Digilent Cmod S7 development board for the FPGA platform.
* Experiment with attacking a suitable target, for example an AVR microcontroller running AES.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 1 February 2024 in Prague.

Bachelor's thesis

ACTIVE NON-INVASIVE ATTACK ON A MICROCONTROLLER BY FAULT INJECTION

Jakub Kučera

Faculty of Information Technology
Department of Information Security
Supervisor: Ing. Jiří Buček, Ph.D.
May 16, 2024

Citation of this thesis: Kučera Jakub. *Active non-invasive attack on a microcontroller by fault injection.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of figures

List of tables

List of algorithms

List of code listings

I would like to thank my supervisor, Ing. Jiří Buček, Ph.D., for his guidance and all the time he devoted to me. I would also like to thank my family for their support.

Declaration

In Prague on May 16, 2024

Abstract

This bachelor's thesis deals with the implementation of a platform for voltage glitching and clock glitching. The platform is used to introduce fault insertion mechanisms with a minimal level of abstraction.

The work contains an implementation of hardware in the Verilog language with firmware written in C and an interface in Python for controlling the platform from a computer. Next, an attack on AES is performed with the implemented platform by injecting faults between the 7th and the 8th MixColumns operation and also between the 8th and the 9th.

The resulting platform is capable of injecting glitches into the power supply as well as a clock signal and also supports the insertion of multiple glitches after one trigger event. The attacks carried out on the AES cipher via fault injection with the implemented platform successfully recovered the secret key.

The contribution of this work is the development of a fault injection tool on the Cmod S7 platform and of a programming interface that enable users to understand the mechanisms of fault injection in detail.

Keywords fault injection tool implementation, fault injection, embedded systems security, AES, FPGA, microcontroller, Verilog, C language, Python

Abstrakt

Tato bakalářská práce se zabývá implementací platformy pro vkládání poruch do napájení a do signálu hodin. Platforma slouží pro představení mechanismu vkládání poruch s minimální úrovní abstrakce.

Práce obsahuje implementaci hardwaru v jazyce Verilog s firmwarem v C a rozhraním pro ovládání z počítače v programovacím jazyce Python. Dále se s implementovanou platformou provádí útok na šifru AES vkládáním poruch mezi 7. a 8. operací MixColumns a také mezi 8. a 9.

Výsledná platforma je schopná vkládat poruchy do napájení i do signálu hodin a také podporuje vkládání více poruch při jednom spouštěcím signálu. Provedené útoky na šifru AES vkládáním poruch s použitím implementované platformy úspěšně odhalily tajný klíč.

Přínosem této práce je vývoj nástroje pro vkládání poruch na platformě Cmod S7 a programovacího rozhraní umožňujících uživatelům detailně pochopit mechanismy vkládání poruch.

Klíčová slova implementace nástroje pro vkládání poruch, vkládání poruch, bezpečnost vestavných systémů, AES, FPGA, mikrontroler, Verilog, jazyk C, Python

List of abbreviations

ADC	Analog-to-Digital Converter
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
DAC	Digital-to-Analog Converter
DFA	Differential Fault Analysis
DFF	D-type Flip Flop
DRAM	Dynamic Random Access Memory
FIA	Fault Injection Attack
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
GPIO	General Purpose I/O
I^2C	Inter-Integrated Circuit
I/O	Input/Output
IP	Intellectual Property
JTAG	Joint Test Action Group
MGIC	Multiple Glitch Insertion Controller
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
PMOD	Peripheral Module
RISC	Reduced Instruction Set Computer
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter

Introduction

In our everyday life, we rely on electronic devices such as computers and microcontrollers. We use them for work, to exchange messages, to watch videos, and more. However, we often forget that security is not given, but it is a never-ending cycle of discovering new attacks and inventing software and hardware protections against them.

In 1997 the first fault injection attack was discovered. This started an endeavor to secure software and microcontrollers against fault injection attacks. Since then, countless other fault injection attacks and techniques have been discovered. To be ahead of adversaries, fault injection tools need to be available for researchers.

This work is intended for those who would like to experiment with voltage glitching and clock glitching attacks and gain knowledge of the inner workings of fault injection platforms. We implement a tool that provides minimal abstraction of injection implementation to facilitate that.

The main goal of this bachelor's thesis is to implement a fault injection platform on an FPGA (Field Programmable Gate Array) that communicates with a target device, a microcontroller, and to perform a fault injection attack with the designed and implemented tool.

The objective of the research part is to study non-invasive fault injection attacks on microcontrollers and fault injection techniques. Next, existing fault injection tools, such as NewAE ChipWhisperer-Nano, are studied.

The aim of the practical part is to design and implement a voltage and clock glitching tool. The tool is an FPGA that drives a crowbar circuit to inject power supply faults. Digilent Cmod S7 development board should be used as the FPGA platform. The practical part optionally includes an implementation of clock glitching. The final goal of the practical part is to perform an attack on a microcontroller. For example, an AVR running an AES cipher.

In chapter 1, we provide the necessary background for understanding this work. We explain what side channels are and how they are relevant to fault injection. We study various non-invasive fault injection techniques and fault injection attacks on microcontrollers. Special attention is paid to a crowbar circuit. We also study existing voltage glitching tools.

The practical part is divided into the design and implementation sections and is further subdivided into hardware, firmware, and software parts. In chapter 2, we describe the design of the FPGA's hardware, firmware, and software. In the following chapter 3, we implement the hardware in Verilog and the firmware in C on top of that. In the same chapter, we also create software in the Python programming language that controls the fault injection tool from a computer.

In chapter 5, we perform an attack on the AES cipher running on an AVR microcontroller using the implemented tool.

Chapter 1

Fault injection

A side channel is an unintended exchange of information between a cryptographic device and its environment. Side channel attacks allow to bypass theoretical safety of cryptographic algorithms by exploiting additional information gained via side channels. Side-channel attacks are implementation-specific and device-specific due to the physical nature of side-channel attacks. [1]

There are many side channels. They include:

- Timing side channel — Non-constant execution time of algorithms can reveal information about handled data or secrets. [2, 1]
- Power side channel — Power consumption of a target device can be measured to gain information about processed input or secret. [3, 1]
- Fault side channel — Consists of injection of faults into a target device and observation of outputs returned by the targeted device. [4, 5]
- Electromagnetic side channel — Electrical devices often generate electromagnetic radiation that can be measured. [6]
- Acoustic side channel — Acoustic emanations, such as keystrokes, can be used by an attacker to obtain passwords or other secrets. [7]

Hardware attacks can be categorized based on two criteria. Firstly, attacks are divided into categories based on their influence on a target device [8]:

- Passive attacks – Such attacks don't interfere with the target device, and only collect data emitted by the target device.
- Active attacks – Active attacks influence the behavior of the target device.
- Passive and active combined attacks – This is a type of attack that combines both approaches. A disturbance of a target device can leave it vulnerable to a passive attack. For instance, one of the proposed passive and active combined attacks combines fault injection with power analysis.

The other possible classification of hardware attacks is based on the required access to the target device [9]:

- Invasive attacks require chip depackaging and passivation layer removal.
- Semi-invasive attacks require chip depackaging, but do not involve passivation layer removal. An example of such an attack is an optical fault induction attack.

- Non-invasive attacks do not involve chip depackaging, but exploit externally observable information such as electromagnetic emissions or power consumption.

Fault injection attacks (FIAs) are active hardware attacks. FIAs introduce faults into running code to reveal secret information by observing the target device's behavior and collecting erroneous outputs. However, some FIAs do not require erroneous outputs to successfully perform the attack. For example, ineffective fault analysis [10] obtains information from fault injections that leave the output unchanged, and safe-error analysis [11] only needs to distinguish between faulted and correct outputs.

Faults can be introduced by various techniques. The techniques range from non-invasive to invasive. Fault injection techniques can be further divided into low-cost and high-cost categories [12]. Low-cost fault injection techniques include underpowering, voltage glitching, clock glitching, device heating, light radiation, etc. Examples of high-cost techniques are the focused light beam, the laser beam, and the focused ion beam.

1.1 Non-invasive fault injection techniques

Non-invasive fault injection techniques are methods of inserting faults into the execution of algorithms without the need to depackage the chips on which the algorithms run.

1.1.1 Clock glitching

Most of the digital circuits are synchronous. In other words, the movement of the data inside them is driven by a clock signal. The clock period cannot be arbitrarily chosen for a circuit, but it is constrained by the following equation [13]:

$$T_{clk} > D_{clk2q} + D_{pMax} + T_{setup} - T_{skew} \tag{1.1}$$

In equation 1.1 T_{clk} denotes the clock period, and D_{pMax} is the maximum propagation time through a combinational logic. The maximum propagation time through a combinational logic is given by a critical path, which is a path with the greatest delay from an input to an output of not necessarily the same register.

"Besides, a precise writing of the timing constraint equation requires taking into account three other parameters: D_{clk2q} delay elapsed between the clock rising edge and the actual update of a register's output; T_{skew} skew or slight phase difference that may exist between the clock signals at the clock inputs of two different registers; T_{setup} setup time which is the amount of time for which a D flip-flop input must be stable before the clock's edge to ensure reliable operation." [13]

Faults induced by a clock glitch are generated by violating the circuit's timing constraints. Since an attacker controls the clock signal, he can decrease the clock period. The decrease of the clock period can result in a metastable behavior due to the setup time violation. During a metastable behavior, an arbitrary value is stored in DFF (D-type Flip Flop). Another option is that the input signal stabilizes at some value that may or may not be the correct one. If the value is not equal to the one achieved during normal operation, a fault occurs. [13]

In another paper, the influence of clock glitches on arithmetical/logical instructions, branch instructions, and memory instructions was observed on an ARM Cortex-M0 with a three-stage pipeline and a two-stage pipeline of an ATxmega256. Insertion of a clock glitch during the fetch stage of an arithmetical/logical or a branch instruction led to the preservation of the previous instruction in a fetch buffer and to the subsequent execution of the same instruction for the second time. A clock glitch inserted during the execution stage of some arithmetical/logical instructions causes the attacked instruction to store incorrect value in a destination register. Memory instructions behaved on the tested platforms differently. When the fetch stage of AVR

ARM Cortex-M0 was attacked, the stored or loaded value was zero. A clock glitch during the execution stage of the ATxmega 256 caused the load instruction to load an arbitrary value. [14]

One disadvantage of clock glitching is that it requires access to the clock signal of the target device. Therefore, this attack cannot be used against devices with internal clock oscillators.

1.1.2 Underpowering

Underpowering is another attack that works on the principle of a timing constraint violation given by equation 1.2. Instead of decreasing the clock period, as is done in clock glitching, the propagation time of signals is increased in the target device's circuit.

For simplicity, the propagation delay T_{PLH} is stated for the rise time of an inverter. In this case is defined as "the difference between the time points at which the input and the output cross $V_{DD}/2$". The propagation delay T_{PLH} is given in equation 1.2. The equation for a high-to-low delay would be similar. [15, 13]

$$T_{PLH} = \frac{C_L \left[\frac{2|V_{TH}|}{V_{DD} - |V_{TH}|} + \ln\left(3 - 4\frac{|V_{TH}|}{V_{DD}}\right)\right]}{\mu_p C_{ox} \frac{W}{L}(V_{DD} - |V_{TH}|)} \tag{1.2}$$

where C_L denotes load capacitance, V_{TH} denotes the threshold voltage of a PMOS, μ_p is the mobility of holes, C_{ox} is the gate capacitance per unit area, W and L respectively denote the width of the PMOS transistor and its length.

Equation 1.2 implies that a decrease in V_{DD} voltage will increase the propagation time in the circuit [15, 13].

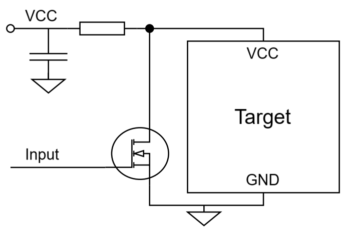
1.1.3 Voltage glitching

Voltage glitching is sometimes referred to as voltage fault injection or power supply glitching. Voltage glitching, unlike underpowering, can be used to target specific instructions by inserting a glitch at a specific time in the program execution. To inject a fault successfully, the right time to insert a glitch must be discovered and a fault-inducing glitch length must also be found. The glitch length must be long enough to induce a fault or multiple faults in an executed code, but it cannot be too long to power down the device.

Voltage glitches can be positive [16] or negative depending on voltage change. Negative voltage glitches decrease the power supply voltage for a short period of time, and positive voltage glitches increase it.

Negative voltage glitches can be inserted with a crowbar circuit [17]. The crowbar circuit consists of an N-Channel MOSFET (Metal Oxide Semiconductor Field Effect Transistor) that shorts the power supply line of a target device to the ground when the MOSFET is driven high. The crowbar circuit is shown in figure 1.1

Figure 1.1 Crowbar circuit. Source: preprint by O'Flynn [17]



Another voltage glitching technique uses an arbitrary waveform generator. Glitches with a generated waveform had a higher success rate in attacks performed on microcontrollers compared to negative voltage glitches inserted with a crowbar circuit. [18]

Zussa et al. [13] carried out an experimental proof of the assumption that negative voltage glitching faults are caused by timing constraint violations by comparing faults obtained by clock glitching and voltage glitching. Because 70% of the time they were identical, they concluded that voltage glitching causes timing constraint violations. However, there might be some spatial effect involved in the case of voltage glitching.

In later work, Zussa et al. [19] observed that positive power supply glitches and negative power supply glitches inject identical faults. Both voltage glitches induce oscillations in the target's power supply, and in both cases negative oscillations cause faults.

Also, it has been theoretically shown that voltage glitches, whose transitions do not occur on clock edges, cannot change the value of a DFF. This was verified by simulation in the same paper. [20]

1.1.4 Electromagnetic fault injection

Electromagnetic fault injection is a technique that does not require physical contact with a target device. Electromagnetic fault injection techniques can be used to perform non-invasive as well as semi-invasive attacks. It was observed that a successful attack is possible on depackaged microcontrollers as well as on capsulated microcontrollers, and that chip packaging did not affect the induced voltage when a fault was injected by a spark gap generator that was more than 10 mm distant from the die's surface of the chip [21].

A possible explanation of how the electromagnetic fault injection technique introduces faults is that it mainly affects the power and ground networks in integrated circuits that have a 3D mesh structure. According to a proposed model, the faults caused by electromagnetic fault injection are sampling faults and not timing faults. This means that the propagation delay is not increased, but rather the circuit's voltage is altered and incorrectly sampled by the DFF. [22]

1.1.5 Heat fault injection

Heating of electronic components is another technique used to induce faults. For instance, a 50 W light bulb was used to inject faults into a DRAM (Dynamic Random Access Memory) by heating it to 100 °C [23]. Another example of a heating attack is an attack on an ATmega162 microcontroller where heating it to 160 °C caused faults in a program execution that led to a successful compromise of a RSA[1] private key [24].

1.1.6 Row hammer attack

Increasing the density of cells in DRAM limits the ability of cells to store a charge and decreases the distance between them. This makes DRAM cells susceptible to disturbances generated by accessing nearby rows of cells. More specifically, a rapid opening and closing of a row accelerates current leakage in nearby rows. The row hammer attack is a software fault injection attack, and no special equipment is needed to carry it out. [25]

1.2 Fault injection attacks

Fault injection attacks can target implementations of cryptographic algorithms such as AES. Various fault injection attacks on the AES have been proposed. Some attack the state of the AES cipher, and others aim to disturb the key expansion algorithm. The attacks also differ in the version of the AES that they target. [26]

[1]RSA is a public-key algorithm developed by Rivest, Shamir and Adelman.

In subsection 1.2.2 a DFA (Differential Fault Analysis) attack [27] is described. In a DFA attack, the attacker compares two outputs encrypted with a cipher with the same plaintext and the same encryption key. If they are different, one of them is assumed to be faulty. Combinations of original and erroneous ciphertexts can leak information about the encryption key. [4]

However, the implementation of encryption algorithms is not the only vulnerable code on microcontrollers. Microcontrollers have debug interfaces that can facilitate, for example, firmware readout. Therefore, it is important to protect debug interfaces of microcontrollers. FIAs can overcome these protections [18].

Nashimoto et al. [28] proposed a hardware/software co-attack. The essence of the attack is to skip instructions that prevent the buffer from overflowing.

1.2.1 Advanced Encryption Standard

This chapter is based on the Advanced Encryption Standard (AES) publication FIPS 197 [29].

AES (Advanced Encryption Standard) is a block cipher. There are three AES ciphers: AES-128, AES-192, and AES-256. The main difference between them is in the key length and the number of rounds, Nr, that are performed. In this work, only AES-128, with a key length of 128 bits and 10 rounds, is discussed and is denoted in the rest of the work as AES.

The input of the AES is a 128-bit key and 128 bits of input data. The output of the algorithm is 128 bits of encrypted input data with the specified key. “Internally, the algorithms for the AES block ciphers are performed on a two-dimensional (four-by-four) array of bytes called the *state*.”[29]. The arrangement of bytes in the array is shown in figure 1.2.

Figure 1.2 AES *state* array input and output. Source: FIPS 197 [29]

$$\begin{array}{cccc} \multicolumn{4}{c}{\textit{input bytes}} \\ \hline in_0 & in_4 & in_8 & in_{12} \\ in_1 & in_5 & in_9 & in_{13} \\ in_2 & in_6 & in_{10} & in_{14} \\ in_3 & in_7 & in_{11} & in_{15} \\ \hline \end{array} \rightarrow \begin{array}{cccc} \multicolumn{4}{c}{\textit{state array}} \\ \hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline \end{array} \rightarrow \begin{array}{cccc} \multicolumn{4}{c}{\textit{output bytes}} \\ \hline out_0 & out_4 & out_8 & out_{12} \\ out_1 & out_5 & out_9 & out_{13} \\ out_2 & out_6 & out_{10} & out_{14} \\ out_3 & out_7 & out_{11} & out_{15} \\ \hline \end{array}$$

Some transformations work with bytes of *state* as if they were elements of Galois field $GF(2^8)$. Byte $b = b_7b_6b_5b_4b_3b_2b_1b_0$ is represented in $GF(2^8)$ as a polynomial $b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$ that is marked as $b(x)$. We also use hexadecimal notation with leading zeros to represent elements of $GF(2^8)$. For example, $x + 1$ is represented as 03.

The addition of two bytes in $GF(2^8)$ is equivalent to performing a *XOR* operation on them, denoted by $\oplus$. The multiplication of bytes b and c consists of their multiplication and modular reduction of the product. The multiplication in $GF(2^8)$ is defined by equation 1.3 and is denoted by the symbol $\bullet$.

$$a(x) \bullet b(x) = a(x)b(x) \bmod x^8 + x^4 + x^3 + x + 1. \tag{1.3}$$

The cipher is composed of rounds during which four byte-oriented transformations transform the *state*: *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. All of the transformations are invertible, so the encrypted text can be decrypted. The order, in which the transformations are called, is displayed in the pseudocode 1.1 where in denotes the input data, Nr represents the number of rounds, and w is the expanded round key. The key expansion is described in the algorithm 1.2

AddRoundKey transformation is a *XOR* operation of the *state* and a round key. The ith round is denoted as K_i. The round keys are obtained by expanding the key that is equal to K_0. The key expansion generates $4 \cdot (Nr + 1)$ four byte words that are denoted as w_i for $0 \leq i < 4 \cdot (Nr + 1)$. The *AddRoundKey* transformation can be written down as equation 1.4.

Algorithm 1.1 AES encryption. Source: FIPS 197 [29], presentation slightly changed

```
1: procedure CIPHER($in, Nr, w$)
2:     $state \leftarrow in$
3:     $state \leftarrow$ AddRoundKey($state, w[0 \ldots 3]$)
4:     for $round$ from 1 to $Nr - 1$ do
5:         $state \leftarrow$ SubBytes($state$)
6:         $state \leftarrow$ ShiftRows($state$)
7:         $state \leftarrow$ MixColumns($state$)
8:         $state \leftarrow$ AddRoundKey($state, w[4 \cdot round \ldots 4 \cdot round + 3]$)
9:     end for
10:    $state \leftarrow$ SubBytes($state$)
11:    $state \leftarrow$ ShiftRows($state$)
12:    $state \leftarrow$ AddRoundeKey($state, w[4 \cdot Nr \ldots 4 \cdot Nr + 3]$)
13:    return $state$
14: end procedure
```

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \oplus \begin{bmatrix} | \\ w_{(4 \cdot round + c)} \\ | \end{bmatrix} \tag{1.4}$$

SubBytes is a non-linear transformation that is applied to each byte of the *state*, and that is composed of two transformations. The first transformation, described in equation 1.5, transforms the byte into its inversion in $GF(2^8)$. The transformation additionally calculates the inverse of zero as zero. The second transformation is a modular reduction given by equation 1.6 in which lower index i denotes the bit of the byte from the least significant bit to the most significant bit.

$$\tilde{x} = \begin{cases} 00 & \text{if } x = 00 \\ x^{-1} & \text{if } x \neq 00 \end{cases} \tag{1.5}$$

$$x'_i = \tilde{x}_i \oplus \tilde{x}_{(i+4) \bmod 8} \oplus \tilde{x}_{(i+5) \bmod 8} \oplus \tilde{x}_{(i+6) \bmod 8} \oplus \tilde{x}_{(i+7) \bmod 8} \oplus b_i \quad \text{where } b = 63 \in GF(2^8) \tag{1.6}$$

Equation 1.6 is written in matrix form in equation 1.7, where subscript i denotes the ith bit starting from the least significant one. $*$ is the multiplication of matrices over GF(2). Later in the work, it is referred to the matrix as a and to the vector as b.

$$\begin{bmatrix} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \\ x'_5 \\ x'_6 \\ x'_7 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} * \begin{bmatrix} \tilde{x}_0 \\ \tilde{x}_1 \\ \tilde{x}_2 \\ \tilde{x}_3 \\ \tilde{x}_4 \\ \tilde{x}_5 \\ \tilde{x}_6 \\ \tilde{x}_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = a * \begin{bmatrix} \tilde{x}_0 \\ \tilde{x}_1 \\ \tilde{x}_2 \\ \tilde{x}_3 \\ \tilde{x}_4 \\ \tilde{x}_5 \\ \tilde{x}_6 \\ \tilde{x}_7 \end{bmatrix} \oplus b \tag{1.7}$$

ShiftRows is a transformation that cyclically shifts every row of the *state* array. Bytes in row r are cyclically shifted r bytes to the left, where r is the index of the row. The transformation is described in equation 1.8.

$$s'_{r,c} = s_{r,(c+r) \bmod 4} \quad \text{for } 0 \leq r < 4 \text{ and } 0 \leq c < 4 \tag{1.8}$$

MixColumns transformation multiplies every column by a matrix. The equation for the transformation is given in equation 1.9.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < 4 \tag{1.9}$$

The round keys of the AES are obtained by expanding the *key* of the AES. The key expansion is described by the algorithm 1.2 where the function *RotWord* rotates a sequence of four bytes to the left by one position, i.e. $\text{RotWord}([x_0, x_1, x_2, x_3]) = [x_1, x_2, x_3, x_0]$. The operation *SubWord* performs a byte-wise replacement defined by the same equation 1.7 as in *SubBytes*. Finally, *Rcon* is an array of constants whose values are shown in table 1.1.

Table 1.1 Rcon array. Source: FIPS 197 [29]

j	$Rcon[j]$	j	$Rcon[j]$
1	[01, 00, 00, 00]	6	[20, 00, 00, 00]
2	[02, 00, 00, 00]	7	[40, 00, 00, 00]
3	[04, 00, 00, 00]	8	[80, 00, 00, 00]
4	[08, 00, 00, 00]	9	[1b, 00, 00, 00]
5	[10, 00, 00, 00]	10	[36, 00, 00, 00]

Algorithm 1.2 AES key expansion algorithm. Source: FIPS 197 [29], presentation slightly changed

```
procedure KeyExpansion($key$)
    $i \leftarrow 0$
    while $i \le Nk - 1$ do
        $w[i] \leftarrow key[4 \cdot i \ldots 4 \cdot i + 3]$
    end while
    while $i \le 4 \cdot Nr + 3$ do
        $temp \leftarrow w[i-1]$
        if $i \bmod Nk = 0$ then
            $temp \leftarrow \text{SubWord}(\text{RotWord}(temp)) \oplus Rcon[i/Nk]$
        else if $Nk > 6$ and $i \bmod Nk = 4$ then
            $temp \leftarrow \text{SubWord}(temp)$
        end if
        $w[i] \leftarrow w[i - Nk] \oplus temp$
        $i \leftarrow i + 1$
    end while
    return $w$
end procedure
```

1.2.2 Attack on AES after the 8th *MixColumns* and before the 9th *MixColumns*

This subsection describes the attack on AES that was proposed by Dussart et al. [30] The aim of the attack is to recover the last round key K_{10}, which is created by the key expansion algorithm. It is possible to perform key expansion in reverse order to recover the AES key from the knowledge of the key K_{10}. The fault model for this attack allows an attacker to insert a random fault into the *state* during the execution of AES. This attack requires about 40 to 50 unique pairs of ciphertexts and faulty ciphertexts.

The attack starts with the injection of a fault between the 8th *MixColumns* and the 9th *MixColumns*. For example, let us assume that a fault is inserted into the first byte $s_{0,0}$ of the AES *state* between the 8th *MixColumns* and the 9th *MixColumns*. The fault caused a one-byte difference that is denoted as ε. After the *ShiftRows* transformation the difference between the unaffected *state* and the erroneous *state* is equal to:

$$\begin{bmatrix} \varepsilon & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 \end{bmatrix} \tag{1.10}$$

The 9th *MixColumns* transformation spreads the error:

$$\begin{bmatrix} 02 \bullet \varepsilon & 00 & 00 & 00 \\ 01 \bullet \varepsilon & 00 & 00 & 00 \\ 01 \bullet \varepsilon & 00 & 00 & 00 \\ 03 \bullet \varepsilon & 00 & 00 & 00 \end{bmatrix} \tag{1.11}$$

AddRoundKey and *SubBytes*, referenced as function $s(x)$, changes the difference between the original *state* and the erroneous *state* to the one in equation 1.12. The $\varepsilon'_0, \varepsilon'_1, \varepsilon'_2$, and ε'_3 represent the differences between the correct ciphertext and the faulty ciphertext at the end of the encryption process. The values of x_0, x_1, x_2, and x_3 represent bytes of the AES state before the last *SubBytes* transformation and are unknown to the attacker.

$$\begin{bmatrix} \varepsilon'_0 & 00 & 00 & 00 \\ \varepsilon'_1 & 00 & 00 & 00 \\ \varepsilon'_2 & 00 & 00 & 00 \\ \varepsilon'_3 & 00 & 00 & 00 \end{bmatrix} \text{ where } \begin{cases} \varepsilon'_0 = s(x_0 \oplus 02 \bullet \varepsilon) \oplus s(x_0) \\ \varepsilon'_1 = s(x_1 \oplus \varepsilon) \oplus s(x_1) \\ \varepsilon'_2 = s(x_2 \oplus \varepsilon) \oplus s(x_2) \\ \varepsilon'_3 = s(x_3 \oplus 03 \bullet \varepsilon) \oplus s(x_3) \end{cases} \tag{1.12}$$

The *ShiftRows* only rearranges the errors:

$$\begin{bmatrix} \varepsilon'_0 & 00 & 00 & 00 \\ 00 & 00 & 00 & \varepsilon'_1 \\ 00 & 00 & \varepsilon'_2 & 00 \\ 00 & \varepsilon'_3 & 00 & 00 \end{bmatrix} \tag{1.13}$$

Finally, the last transformation *AddRoundKey* does not affect the fault matrix. Therefore, the difference between the original *state* and the *state* with an inserted fault is given by equation 1.13.

Information about the last round key K_{10} can be gained from the last *SubBytes* transformation by solving a system of equations 1.14, where $x_0, x_1, x_2, x_3, \varepsilon$ are unknown variables that represent bytes of the AES *state* before the last *SubBytes* transformation.

$$\begin{cases} \varepsilon'_0 = s(x_0 \oplus 02 \bullet \varepsilon) \oplus s(x_0) \\ \varepsilon'_1 = s(x_1 \oplus \varepsilon) \oplus s(x_1) \\ \varepsilon'_2 = s(x_2 \oplus \varepsilon) \oplus s(x_2) \\ \varepsilon'_3 = s(x_3 \oplus 03 \bullet \varepsilon) \oplus s(x_3) \end{cases} \tag{1.14}$$

The system of equations 1.14 can be generalized as a single equation 1.15.

$$\varepsilon' = s(x \oplus c \bullet \varepsilon) \oplus s(x) \text{ where } c \in \{01, 02, 03\} \tag{1.15}$$

It is possible to calculate a set of possible values of the inserted fault and four bytes of the key K_{10} from the equation system in equation 1.14. Recovery of the key K_{10} consists of three steps. First, a set of possible values of ε is calculated. Then, a set of possible values of the AES *state* is obtained. Finally, a set of possible values of a key K_{10} is calculated.

A set of possible values of the injected fault ε is calculated for every valid combination of the differential fault ε' and the coefficient c. The set of possible values of the injected fault ε for a specific ε and c is denoted as $E_{c,\varepsilon'}$. The formula for $E_{c,\varepsilon'}$ is written in equation 1.16.

$$E_{c,\varepsilon'} = \{\varepsilon \in GF(2^8) : \exists x \in GF(2^8), \varepsilon' = s(x \oplus c \bullet \varepsilon) \oplus s(x)\} \tag{1.16}$$

Then, a new set E is calculated that is an intersection of four $E_{c,\varepsilon'}$ sets. For example, with the same fault location as in the fault propagation example, E would be as shown in equation 1.17.

$$E = E_{2,\varepsilon'_0} \cap E_{1,\varepsilon'_1} \cap E_{1,\varepsilon'_2} \cap E_{3,\varepsilon'_3} \tag{1.17}$$

The values of the E set are used to calculate the possible values of bytes of the AES *state* before the last *SubBytes* transformation. For every fault $e \in E$, equation 1.18 is solved with the same combinations of c and ε' as in the last step. Equation 1.18 can be obtained from equation 1.15 by writing $s(x)$ as $a * x^{-1} \oplus b$, which is the definition of *SubBytes* transformation, and performing a few operations so that the equality still hold true.

$$x^2 \bullet (c \bullet \varepsilon)^{-2} \oplus x \bullet (c \bullet \varepsilon)^{-1} = (a^{-1} * \varepsilon')^{-1} \bullet (c \bullet \varepsilon)^{-1} \tag{1.18}$$

By substituting $x \bullet (c \bullet \varepsilon)^{-1}$ for t and the right-hand side for θ, we obtain equation 1.19.

$$t^2 \oplus t = \theta \tag{1.19}$$

Equation 1.19 has two solutions: α and β. We get $y_1 = \alpha \bullet c \bullet \varepsilon$ and $y_2 = \beta \bullet c \bullet \varepsilon$ after undoing the substitution for $x \bullet (c \bullet \varepsilon)^{-1}$. There are two more solutions $y_3 = 0$ and $y_4 = c \bullet \varepsilon$, if $\theta = 01$.

The possible values of the last round key K_{10} are obtained by calculating $K_{10}[i] = s(y_k) \oplus C'[i]$, where C' is the erroneous ciphertext, $k \in \{1, 2\}$ or $k \in \{1, 2, 3, 4\}$ based on the value θ, and i is the index of a key byte that is being calculated. The XOR operation eliminates the value of the state before the *AddRoundKey* operation and reveals a possible byte of key K_{10}.

The calculations are repeated for other combinations of coefficients c, corresponding ε' differential faults and injected fault values ε from set E and with different ciphertext pairs, until only one possible value of the four bytes of the key K_{10} is left. This process can be replicated for the remaining twelve K_{10}'s bytes to recover the whole last round key K_{10}.

The algorithm 1.3 describes how to obtain four bytes of the AES' last round key K_{10} using the steps described above. The algorithm takes as input an array of correct ciphertexts *cts*, an array of corresponding erroneous ciphertexts $faulty\_cts$, into which a fault has always been injected into the same column of *state*, and an array of indices $fault\_idx$, which indicate the faulty bytes, for example $[0, 13, 10, 7]$.

Once the key K_{10} is recovered, the inverse key expansion algorithm 1.4 is performed to gain the key K_0, i.e. the AES encryption key.

1.2.3 Attacks on AES after the 7th *MixColumns* and before the 8th *MixColumns*

Dussart et al. [30] also presented a fault injection attack on AES that requires only ten faults. The main idea is that the fault inserted between the 7th and 8th *MixColumns* is transformed into four faults after the 8th *MixColumns*. Then the key K_{10} is calculated from the faults in the same way as in the attack described in the previous subsection. The only difference is that the attack is carried out for all four columns of every ciphertext. This is the reason why only ten ciphertext pairs are needed.

Algorithm 1.3 Recovery of K_{10} with fault between the 8th and the 9th *MixColumns*. Source: article by Dusart et al. [30], edited into pseudocode

```
1:  procedure GetFaultValues($c$, $\varepsilon'$)
                ▷ Faster way to calculate $\{\varepsilon \in GF(2^8) : \exists x \in GF(2^8), \varepsilon' = s(x \oplus c \bullet \varepsilon) \oplus s(x)\}$
2:      $\varepsilon\_set \leftarrow \{\}$
3:      for $t$ in $\{01, \dots, 1F, 40, \dots, 5F, A0, \dots, BF, E0, \dots, FF\}$ do
4:          $\varepsilon\_set \leftarrow \varepsilon\_set \cup (c \bullet (a^{-1} \bullet \varepsilon') \bullet t)^{-1}$
5:      end for
6:  end procedure

7:  procedure GetKeyValues($c$, $\varepsilon$, $\varepsilon'$, $fault$)
8:      $\theta \leftarrow ((a^{-1} * \varepsilon') \bullet c \bullet \varepsilon)^{-1}$        ▷ $a$ is the matrix from SubBytes definition
9:      $\alpha \leftarrow$ solution of $t^2 \oplus t = \theta$
10:     $\beta \leftarrow \alpha \oplus 01$
11:     if $\theta \neq 01$ then
12:         return $\{s(c \bullet \varepsilon \bullet \alpha) \oplus fault, s(c \bullet \varepsilon \bullet \beta) \oplus fault\}$
13:     else
14:         return $\{s(c \bullet \varepsilon \bullet \alpha) \oplus fault, s(c \bullet \varepsilon \bullet \beta) \oplus fault, b \oplus fault, s(c \bullet \varepsilon) \oplus fault\}$
15:     end if        ▷ $b$ on the line above is the constant from SubBytes definition
16: end procedure

17: procedure Recover89($cts$,$faulty\_cts$,$fault\_idx$)
18:     for $i$ in $fault\_idx$ do        ▷ Possible $K_{10}$'s bytes that match all ciphertext pairs
19:         $keys[i] \leftarrow \{00, \dots, FF\}$
20:     end for
21:     for $ct, faulty\_ct$ in $cts$, $faulty\_cts$ do        ▷ For every pair of correct and faulty ciphertext
22:         for $i$ in $fault\_idx$ do        ▷ Possible $K_{10}$'s bytes from a single fault
23:             $keys\_single\_set[i] \leftarrow \{\}$
24:         end for
25:         for $coeffs$ in $[[02, 01, 01, 03], [03, 02, 01, 01], [01, 03, 02, 01], [01, 01, 03, 02]]$ do
                        ▷ Coefficients depend on the fault's position in the state column
26:             $\varepsilon\_set \leftarrow \{00, \dots, FF\}$        ▷ Set of possible values of the injected fault
27:             for $c, i$ in $coeffs, fault\_idx$ do
28:                 $\varepsilon' \leftarrow faulty\_ct[i] \oplus ct[i]$        ▷ Difference between the ciphertexts
29:                 $\varepsilon\_set \leftarrow \varepsilon\_set \cap$ GetFaultValues($c, \varepsilon'$)
30:             end for
31:             for $c, i$ in $coeffs, fault\_idx$ do        ▷ Possible key values based on $\varepsilon$
32:                 $\varepsilon' \leftarrow faulty\_ct[i] \oplus ct[i]$        ▷ Difference between the ciphertexts
33:                 $fault \leftarrow faulty\_ct[i]$        ▷ Value of the faulty byte in the output of AES
34:                 for $\varepsilon$ in $\varepsilon\_set$ do
35:                     $keys\_single\_set[i] \leftarrow keys\_single\_set[i] \cup$ GetKeyValues($c, \varepsilon, \varepsilon', fault$)
36:                 end for
37:             end for
38:         end for
39:         for $i$ in $fault\_idx$ do
40:             $keys[i] \leftarrow keys[i] \cap keys\_single\_set[i]$
41:         end for
42:     end for
43:     return $keys$
44: end procedure
```

Algorithm 1.4 Inverse key expansion algorithm. Source: preprint by Dusart et al. [31]

```
procedure InverseKeyExpansion($K_{10}$)
    $i \leftarrow Nb \cdot (Nr + 1) - 1$
    $j \leftarrow Nk - 1$
    while $j \geq 0$ do
        $w[i] \leftarrow K_{10}[4 \cdot j \ldots 4 \cdot j + 3]$
        $i \leftarrow i - 1$
        $j \leftarrow j - 1$
    end while
    while $i \geq 0$ do
        $temp \leftarrow w[i + Nk - 1]$
        if $i \bmod Nk = 0$ then
            $temp \leftarrow \text{SubWord}(\text{RotWord}(temp)) \oplus Rcon[i/Nk + 1]$
        else if $Nk > 6$ and $i \bmod Nk = 4$ then
            $temp \leftarrow \text{SubWord}(temp)$
        end if
        $w[i] \leftarrow w[i + Nk] \oplus temp$
        $i \leftarrow i - 1$
    end while
    return $w$
end procedure
```

To update the algorithm 1.3 to perform this improved variant of the attack, the lines 22-41 have to be nested into another for loop. The for loop iterates through every combination of four faulty bytes, i.e. $[[0, 13, 10, 7], [4, 1, 14, 11], [8, 5, 2, 15], [12, 9, 6, 3]]$, that are assigned to $faulty\_idx$ variable. Also, the code on the line 18 has to initialize all 16 sets. After the modifications, the algorithm returns the whole K_{10} instead of four of its bytes.

Piret and Quisquater [32] proposed an attack on AES that needs only two erroneous ciphertexts. Based on their observation, two ciphertexts are sufficient in 77% of the cases. The proposed attack applies to ciphers with a "Substitution-Permutation structure" [32] such as AES. These ciphers can be described by equation 1.20. The last round of the cipher does not contain the θ layer.

$$\sigma[K_{Nr}] \circ \gamma_{N}r \circ (\circ_{r=1}^{Nr-1} \sigma[K_r] \circ \theta_r \circ \gamma_r) \circ \sigma[K_0] \text{ where} \tag{1.20}$$

- Nr denotes the number of cipher's rounds.
- "γ_r layer consists in the parallel application of n 8×8 S-boxes (not necessarily identical)" [32].
- σ_r is a key addition layer $\sigma[k](a) = b \Leftrightarrow b_j = a_j \oplus k_j, 1 \leq j \leq n$.

In the case of the AES, the θ layer is represented by the *ShiftRows* and *MixColumns* transformations, and the γ layer consists of the *SubBytes* operation. Because the additional *ShiftRows* transformation "has no cryptographic significance" [32], the attack is applicable to AES.

The main idea behind the attack is to check for all possible values of K_{10}, if the difference between the correct ciphertext C and the erroneous ciphertext C' could be generated by a single-byte fault transformed by the θ layer. The equation is given in equation 1.21, where D is a 16-byte array with a single non-zero byte that represents the single-byte fault. [32]

$$\theta(D) = \gamma_{Nr}^{-1} \circ \sigma[K_{Nr}](C) \oplus \gamma_{Nr}^{-1} \circ \sigma[K_{Nr}](C') \tag{1.21}$$

Again, the equation can be written as equation 1.22 in the case of the AES. Because the AES has *ShiftRows* operation after the last *MixColumns*, the bytes of the ciphertext pairs and of K_{10} must be reordered accordingly.

$$\text{MixColumns}(\text{ShiftRows}(D)) = s^{-1}(K_{10} \oplus C) \oplus s^{-1}(K_{10} \oplus C') \tag{1.22}$$

The attack consists of four phases. In the first phase, faults are injected between the 7th and the 8th *MixColumns* of the AES. Then, the key space is reduced by finding possible values of the key K_{10} byte by byte, if equation 1.22 is satisfied for selected bytes for two pairs of correct and faulty ciphertexts. All obtained values of the key K_{10} are tested, whether they satisfy the same conditions as in the search space reduction step, but this time only for all bytes at once. Keys that fail the test are no longer considered. This is repeated until only one candidate for K_{10} is left. Finally, an inverse key expansion is performed to recover the AES encryption key. [32]

The algorithm for steps two and three is located in appendix A due to its length. Instead, the algorithm 1.5 shows a simplified version [33] of the attack that does not have the key space reduction step.

The algorithm 1.5 solves the system of equations 1.23 [33]. The system of equations 1.23 is from article [26].

The system of equations 1.23 is similar to equation 1.22. The difference is that the system of equations 1.23 contains an equation only for four bytes of the last round key K_{10}. Therefore, K_{10} is recovered four bytes at a time.

$$\begin{cases} 2 \bullet \varepsilon = s^{-1}(C_{0,0} \oplus K_{10,0,0}) \oplus s^{-1}(C'_{0,0} \oplus K_{10,0,0}) \\ \varepsilon = s^{-1}(C_{1,3} \oplus K_{10,1,3}) \oplus s^{-1}(C'_{1,3} \oplus K_{10,1,3}) \\ \varepsilon = s^{-1}(C_{2,2} \oplus K_{10,2,2}) \oplus s^{-1}(C'_{2,2} \oplus K_{10,2,2}) \\ 3 \bullet \varepsilon = s^{-1}(C_{3,1} \oplus K_{10,3,1}) \oplus s^{-1}(C'_{3,1} \oplus K_{10,3,1}) \end{cases} \tag{1.23}$$

Algorithm 1.5 takes as an input an array of correct ciphertexts *cts*, and an array of corresponding erroneous ciphertexts *faulty_cts* that were obtained by injecting a fault between the 7th and the 8th *MixColumns* transformation. It returns the key K_{10}.

1.2.4 Attacks on debug interface protection of microcontrollers

One of the microcontroller's attack vectors is the debug interface. A debug interface enables, for example, firmware programming and firmware reading. Therefore, it is necessary to secure these interfaces. [18]

There are many different types of microcontrollers made by various manufacturers. Each microcontroller might have its own debug interface protection mechanism. Due to this, attacks on microcontrollers do not apply to all of them. This subsection describes a few security mechanisms that can be defeated with voltage glitching.

STM32F103 microcontroller includes a *Readout Protect* command that enables the read protection feature. When an STM32F103 receives the *Readout Protect* command, it checks a readout protection value before sending back a part of the firmware. This check can be overcome by injecting a glitch during the check. [18]

Wiersma and Pareja [34] used voltage glitching to defeat different debug interface protections. A different type of microcontroller protects the debug interface with a 128-bit password. Such protections can be overcome with voltage glitching. Attack on another microcontroller tried to change a JTAG configuration fetched from Read-Only memory via voltage glitching. The debug interface of that microcontroller could also be unlocked by affecting the fetched value.

1.2.5 Buffer overflow attacks with fault injection

Nashimoto et al. [28] proposed an attack that combines software and hardware attacks. They demonstrated that the attack defeats input size limitation protections, i.e. functions that can

Algorithm 1.5 Simplified recovery of K_{10} with fault between the 7th and the 8th *MixColumns*. Source: tutorial by Shchavleva [33], modified for fault between the 7th and the 8th *MixColumns*

```
procedure RECOVER78SIMPLE($cts$, $faulty\_cts$)
    $keys \leftarrow \{[]\}$                                   ▷ Possible values of $K_{10}$
    for $mask$ in $[[0,13,10,7],[4,1,14,11],[8,5,2,15],[12,9,6,3]]$ do
                                  ▷ For indices of every column after ShiftRows
        for $ct, faulty\_ct$ in $cts, faulty\_cts$ do
            $key\_set \leftarrow [\{\},\{\},\{\},\{\}]$             ▷ For every ciphertext pair
            for $\varepsilon$ in $\{00,\dots,FF\}$ do      ▷ For possible faults that affected the column
                for $coeffs$ in $[[02,01,01,03],[03,02,01,01],[01,03,02,01],[01,01,03,02]]$ do
                    for $i$ in $\{0,1,2,3\}$ do
                        $c \leftarrow coeffs[i]$
                        $ct\_byte \leftarrow ct[mask[i]]$
                        $faulty\_byte \leftarrow faulty\_ct[mask[i]]$
                        $key\_byte \leftarrow \{k \in GF(2^8) : \varepsilon \bullet c = s^{-1}(k \oplus faulty\_byte) \oplus s^{-1}(k \oplus ct\_byte)\}$
                        $key\_set[i] \leftarrow key\_set[i] \cup key\_byte$
                    end for
                end for
            end for
            $keys \leftarrow [key, k_0, k_1, k_2, k_3] : key \in keys, k_j \in key\_set[j], j \in 0,\dots,3$
                  ▷ Extend values in keys by all combinations of key's bytes in key_set
        end for
    end for
    return ShiftRows($keys[0]$)                  ▷ Arrange the bytes in the correct order
end procedure
```

limit the input size. An example of such function is `strncpy()`.

Nashimoto et al. used clock glitching fault injection technique to insert faults into `strncpy()` function that was running on an 8-bit ATmega163. They were targeting instruction in a for loop that held the number of bytes to be written into a buffer. By skipping the decrement operation of the mentioned variable five times, they were able to write five extra bytes out of the buffer. The caused buffer overflow allowed them to hijack the program flow. They also successfully performed the attack on a 32-bit ARM Cortex-M0+ microcontroller. [28]

1.3 Hardware prerequisites

In this section, we briefly describe some of the hardware terms that are mentioned later in this work.

Field Programmable Gate Array (FPGA) is a semiconductor device that can be reprogrammed after manufacturing. This is possible because it consists of configurable logic blocks and programmable interconnects.

Block RAM (BRAM) is a type of memory resource in FPGAs. BRAM is a large chunk of memory that is suitable for storing a few kilobytes of data. [35]

Advanced Microcontroller Bus Architecture Advanced eXtensible Interface (AMBA AXI) is a transaction-based protocol with independent read and write channels. [36]

Serial Serial is an interface that transfers bits in series, or in other words, bit by bit. [37]

Universal Asynchronous Receiver Transmitter (UART) is an integrated circuit that handles asynchronous transmission. “Asynchronous serial data is usually sent one binary word at a time. Each is accompanied by start and stop bits that define the beginning and end of a word”. [37]

Serial Peripheral Interface (SPI) uses three wires for communication. Two of them are used for data transmission, and the last determines with which slave is a master communicating. [37]

I^2C (I^2C) is two wire interface. One wire is used for the clock signal and the other is used for sending data. [37]

Joint Test Action Group (JTAG) interface is used for circuit testing. It contains five signals, two of which are used for data transmission. [37]

Smart card interfaces can be contactless or with contacts. Smart cards might have an ISO 7816 interface with contacts or a contactless ISO 14443 interface. [38]

1.4 Existing fault injection solutions

Here is given a brief overview of how a typical voltage and clock fault injection tool functions. Some fault injection techniques, such as clock glitching or voltage glitching, require that the glitch is inserted at a specific time in the execution of a program. One possible way to insert a glitch at a specific time in the code execution is to wait for the reception of a trigger event. For example, a logic level change of the target device’s output or a specific character received via a communication interface. Then a given number of target’s clock cycles and an optional finer offset later a glitch is inserted.

Another important factor is the duration of the glitch. Sometimes multiple glitches have to be inserted for a successful fault injection attack. In that case, it is desirable to be able to control each delay between inserted glitches and the length of each glitch individually.

A comparison of fault injection capabilities of various solutions can be done based on the variety and flexibility of trigger mechanisms, glitch offset range and resolution, glitch width range and resolution, a maximum count of inserted glitches and a voltage fault injection mechanism. Then there are features for controlling a target device. These can include, but are not limited to, target clock frequency generation range, communication with the target device, and ability to program the target device.

1.4.1 NewAE ChipWhisperer

O’Flynn created an open source side-channel analysis platform that is capable of clock glitching and power trace capture. The original ChipWhisperer’s reference implementation runs on a ZTEX Spartan 6 LX25 FPGA Module, but it can be programmed into the control FPGA on SAKURA-G or SASEBO-W platform. The ChipWhisperer’s FPGA design is modular. Modularity is achieved by connecting all blocks, such as a glitch generator or a trigger unit, to a central register controller. [17]

The original ChipWhisperer generates clock glitches by performing XOR or OR logical operation of the clock signal with a glitch signal. The glitch width is controlled by shifting the phase of the glitch signal and changing its delay. The original ChipWhisperer tool cannot insert voltage glitches. [17]

O’Flynn founded in 2013 a NewAE Technology Inc. company [39] that manufactures a product line of side-channel power analysis and fault injection devices named ChipWhisperer. The company designed four ChipWhisperer devices: ChipWhisperer-Nano, ChipWhisperer-Lite, ChipWhisperer-Pro, and ChipWhisperer-Husky. These tools are capable of voltage glitching and power trace capture, and all of them, except for the ChipWhisperer-Nano, are also capable of clock glitching. The voltage glitches are inserted with a crowbar circuit on all of the platforms. [40]

ChipWhisperer-Nano is a low-cost platform that is mainly intended for side-channel power analysis. The platform is based on a microcontroller that handles communication with a target device and voltage glitch insertion. The microcontroller is used instead of an FPGA to reduce the cost of the platform. The resolution of glitch width and glitch offset is about 8.3 ns, but it suffers with high jitter. The ChipWhisperer-Nano is not capable of measuring the offset in terms of the target's clock cycles. [40, 41]

Both ChipWhisperer-Lite and ChipWhisperer-Pro upgraded their platform from a microcontroller to an FPGA. That allows them to generate almost any target clock frequency, measure glitch offset in the target's clock cycles, and perform voltage and clock glitching with sub 1 ns resolution. [40, 42, 43]

ChipWhisperer-Husky is the successor to ChipWhisperer-Lite. The improvements include a higher resolution of the glitch offset and the glitch width. The achievable glitch resolution is 833 ps. ChipWhisperer-Husky also introduces an option to insert multiple independent glitches. [44]

1.4.2 Tools by Riscure

Riscure is another company that provides voltage glitching tools. Their FPGA-based tool named Spider is capable of generating arbitrary waveforms with 4 ns resolution. The spider supports SPI, JTAG, I^2C, and UART protocols, and has an SDK that supports Python, Java, and C. [45]

Another tool offered by Riscure is a VC glitcher. The VC glitcher is built on an FPGA as well. The key feature of the VC glitcher is glitch insertion with a programmable amplitude and duration. Their tool is capable of generating voltage patterns with 2 ns resolution and 500 samples in length. [46]

1.4.3 Generic Implementation Analysis Toolkit

This section is about GIAnT (Generic Implementation Analysis Toolkit) presented in an article [47] and later improved as a part of a Ph.D. thesis [48]. GIAnT is a low-cost platform for performing non-invasive fault injection. GIAnT is an open source platform that is built around Spartan6 FPGA. The platform is capable of manipulating the target's supply voltage with a 16-bit DAC (Digital-to-Analog Converter) with a 10 ns resolution, measuring the target's power consumption with an ADC (Analog-to-Digital Converter) with the same maximum sample rate of 100 MHz and triggering the glitch by detecting patterns in the target's power consumption.

GIAnT can communicate with smartcards via an ISO 14443 reader, an ISO 7816-compliant smartcard interface, and with other targets via GPIO (General Purpose I/O) pins. GIAnT communicates with the PC via USB with the help of a microcontroller, which is part of the platform. GIAnT can be controlled with an API (Application Programming Interface) that is implemented in C++.

GIAnT is capable of injecting faults with various fault injection techniques. It can perform optical fault injection with a modified electronic flash of a photo-camera. The flash can be replaced with a coil to inject faults by generating an electromagnetic field.

Another type of attack that GIAnT is capable of is clock glitching. The platform inserts clock glitches by generating two clock signals with the second one being slightly offset from the first one. Then a logical AND or a logical OR operation is performed on both clock signals to shorten or lengthen the output clock signal. The operations on the clock signal are performed with external circuitry outside of the FPGA. The resolution of a clock glitch is 1/256th of a clock period, which is 10 ns.

1.4.4 µ-Glitch hardware framework

Saß et al. [49] present a hardware framework named µ-Glitch capable of injecting multiple consecutive voltage faults. Their framework consists of "the Clock Generation Unit, the Host Communication Unit, the I/O Buffer Unit, internal Configuration Registers, the Multiple Voltage Fault Unit and the Serial Target I/O Unit" [49].

Their Multiple Fault Injection Unit consists of multiple Single Fault Units. Single Fault Units take as an input clock, trigger, offset and width, and they have two outputs: Fault Done and Fault Out. The former is asserted for one clock cycle after executing a fault attempt. The latter indicates that a glitch should be inserted. [49]

In the Multiple Fault Injection Unit, Single Fault Units are chained together to provide the capability of inserting multiple faults triggered by a single trigger event. The chaining is done in such a way that the Fault Done signal of the previous Single Fault Unit is connected to the trigger input of the next Single Fault Unit in the chain via a demultiplexer. Demultiplexers are used to control the count of inserted faults. The demultiplexers pass the Fault Done signal to another Single Fault Unit in the chain or interrupt the chain by not passing the Fault Done to the next Single Fault Unit. If a demultiplexer interrupts the chain, it signals the end of all fault injections instead. Fault Out signals are connected with OR logic to provide one output that activates the crowbar circuit. [49]

1.4.5 Summary

The capabilities of each platform are summarized in table 1.2

Table 1.2 Capabilities of fault injection tools

Platform name	Offset resolution	Glitch resolution	Glitch types
CW-Nano [50]	8.3 ns (high jitter)	8.3 ns (high jitter)	Crowbar
CW-Lite [50]	0.4% of clock cycle	0.4% of clock cycle	Crowbar, clock
CW-Pro [50]	0.4% of clock cycle	0.4% of clock cycle	Crowbar, clock
CW-Husky [50]	0.833–1.666 ns	0.833–1.666 ns	Crowbar, clock
Spider [51]	N/A	4 ns	Voltage waveform
VC glitcher [46]	N/A	2 ns	Voltage waveform, clock
µ-Glitch [49]	N/A	N/A	Crowbar
GIAnT [48, 47]	10 ns	10 ns (voltage) 0.4% (clock)	Voltage waveform, clock, optical, electromagnetic

Platform name	Multiple fault injection	Target clock frequency
CW-Nano [50]	No	3.75, 7.5, 15, 30 or 60 MHz
CW-Lite [50]	Consecutive clock cycles	5–200 MHz
CW-Pro [50]	Consecutive clock cycles	5–200 MHz
CW-Husky [52]	Yes, same glitch type	10–350 MHz
Spider [45]	N/A	N/A
VC glitcher [46]	N/A	N/A
µ-Glitch [49]	Yes	N/A
GIAnT [47]	Yes, (different glitch types — N/A)	N/A

Platform name	Trigger types
CW-Nano [50]	Rising edge
CW-Lite [50]	Rising edge
CW-Pro [50]	Rising edge, analog pattern, UART, SPI
CW-Husky [50]	Edge/level, analog pattern and threshold, UART, ARM trace
Spider [45]	N/A
VC glitcher [46]	N/A
µ-Glitch [49]	N/A
GIAnT [47]	Rising edge

Platform name	Programming API	Communication with target
CW-Nano [53, 50]	Python	UART
CW-Lite [53, 50]	Python	UART
CW-Pro [53, 50]	Python	UART
CW-Husky [53, 50]	Python	UART
Spider [45]	Python, Java, C	SPI, JTAG, I^{2}C, UART
VC glitcher [46]	N/A	Smart card connector
µ-Glitch [49]	N/A	Serial
GIAnT [47]	C++	Serial, ISO 7816, ISO 14443

Chapter 2
Design

In this chapter, we design the fault injection platform based on Cmod S7. We took into consideration the features of the Cmod S7 during the design process of the fault injection platform.

Cmod S7 is a board with an FPGA designed for use with breadboards because it has 32 digital I/O pins at the bottom of the board. The FPGA on Cmod S7 is Xilinx Spartan-7. It is also equipped with a PMOD (Peripheral Module) connector, which has 8 additional I/O pins and 2 pairs of pins dedicated to ground and power delivery. Another feature of the CMod S7 is a USB-UART bridge, which can be used for communication with a computer. [54]

This chapter is divided into three parts. First, we focus on the hardware design that is implemented using the flexible logic inside of the FPGA. Then, we design the firmware that controls the hardware. Lastly, we design an interface that runs on the computer. The interface allows users of the platform to control the glitch parameters and retrieve the results in a simple way.

2.1 Hardware design

The foundation of every platform is a well-designed hardware. The functional requirements for the hardware are the following:

- Capability to insert voltage glitches.
- Ability to insert clock glitches.
- Controllable glitch offset and delay.
- Controllable glitch width.
- Capability to trigger glitch insertion.
- Generation of a clock for a target.
- Interface for communication with a target device.
- Adjustable glitch parameters from a computer.

2.1.1 Voltage glitch insertion

One of the requirements is the voltage glitching capability. The voltage glitch is inserted via a crowbar circuit. The crowbar circuit can be driven by a single signal. Therefore, we need to design a mechanism to control the glitch signal. The signal can be set to high to activate a glitch by driving a transistor in the crowbar circuit, or set to low to not insert a glitch.

The glitch insertion is also controlled by offset, delay, and width of the glitch. These parameters are described in the following subsections.

2.1.2 Clock glitch insertion

The clock has its own glitch insertion signal that can be enabled independently of the voltage glitch insertion signal, but it is controlled by the same offset, delay, and width parameters. When the clock glitch is enabled, it flips the value of the target clock signal by using the XOR operation on the target clock signal.

2.1.3 Glitch offset

To be able to insert glitches precisely at the desired time, we decided to use two parameters: glitch offset and glitch delay. The two parameters are used to enable long delays between a trigger and an injected glitch and also to ease the synchronization of the glitch insertion with the target's clock for the user of the platform.

The glitch offset parameter determines how many target clock cycles it takes to insert a glitch. The glitch offset is also synchronized with the target's clock. That means that a fault injection without using a glitch delay always activates at the rising edge of the target's clock. This is especially necessary for clock glitching, where it is desirable to have control over the glitch's position relative to a target clock signal.

The delay parameter is implemented in the same way as the glitch width, and for that reason, it is described in the subsection 2.1.4 together with the glitch width.

2.1.4 Glitch width and glitch delay

We designed the hardware so that the glitch width and the glitch delay share the same hardware. What differs between a glitch delay and a glitch width is another parameter named glitch type. The glitch width and glitch delay parameters are stored as an array of integers. An additional array of glitch-type variables specifies whether a glitch should be inserted and eventually what type of glitch should be inserted. To unify the naming of the glitch widths and delays in the first array, we call the array glitch duration array and the values in it glitch duration.

Storing both width and delay as a single variable together with a glitch type in a glitch duration array has also an added benefit. By increasing the array size to more than two, our design is capable of inserting multiple glitches in succession. For example, a multiple glitch configuration would contain glitch durations in one array and alternating glitch-on and glitch-off values in the second array.

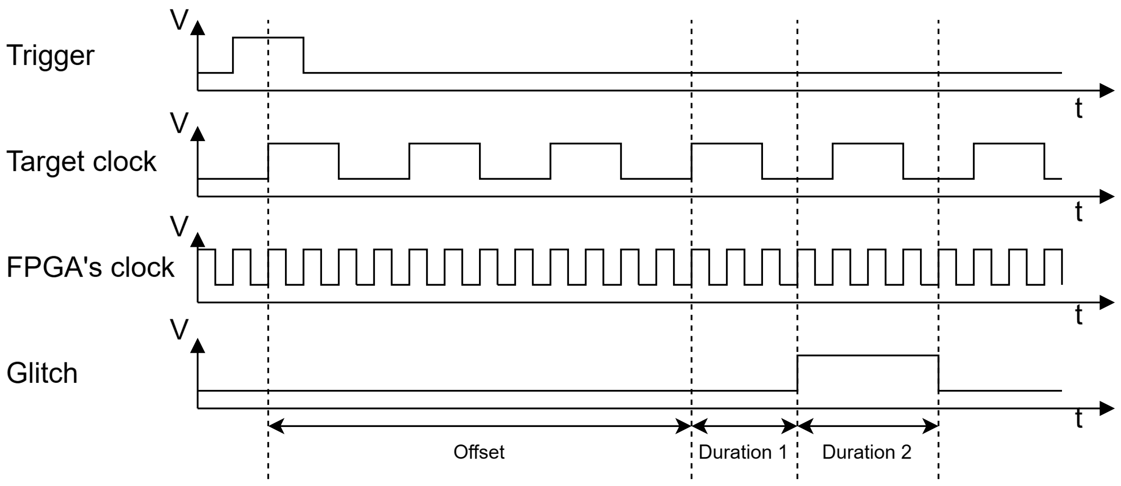
The glitch type can also specify that both clock glitching and voltage glitching should be active at the same time.

Figure 2.1 shows the difference between the glitch offset and the glitch duration. The resolution of the offset is in target clock cycles and the resolution of the durations is in the FPGA's clock cycles. Alternatively, duration 1 and duration 2 could be denoted as the glitch delay and the glitch width, respectively.

2.1.5 Target clock generation

The target clock is generated from the FPGA's clock by dividing it. Doing it this way ensures that its rising edges are synchronized with the FPGA's clock. The downside of this design is that the target clock frequency generator does not support fractional divisors. If we allow the

Figure 2.1 Difference between glitch offset and glitch duration



generation of an arbitrary clock frequency of the target clock signal, the glitches would not always be inserted at the same position in the target clock cycle.

2.1.6 Glitch trigger

The glitch is generally inserted relative to some reference point. The reference point can be an event in the target device, such as a rising edge of an output signal.

We designed three trigger options:

- Falling edge of the target reset signal.
- Rising edge of the external trigger signal.
- Manual trigger.

The falling edge of the target reset signal is useful in situations when the target runs an algorithm after it is reset. The rising edge of an external trigger can be used in instances that do not involve resetting the target device. Finally, the primary purpose of the manual trigger is for testing.

The external trigger goes through two registers in a series before its input is used in further logic. This is because of a clock domain crossing that exists between the FPGA's clock and the microcontroller's clock.

2.1.7 Communication interfaces

We chose the UART interface as the communication interface between the FPGA and the PC since the CMod S7 comes with a USB-UART bridge.

For data transfer between the FPGA and the target, we also used a UART interface. Hence, the FPGA design contains two UART interfaces.

2.1.8 Adjustable glitch parameters

To make the glitch parameters configurable, the glitch settings must be stored in configuration registers, which are accessible from a PC. In our design, a microprocessor receives commands from the USB-UART connection to the PC and then configures the configuration registers via AXI.

The AXI is inserted between the microprocessor and the configuration registers together with the fault injection hardware to make the glitch insertion circuitry independent from other components of the fault injection platform, i.e. the microprocessor and the communication interfaces. We selected the AXI because of the support provided by the tools, which we name in the implementation section.

2.2 Firmware design

Because our design uses a microprocessor to transfer data between the UART interfaces and the fault injection peripheral, we need to design firmware for the microprocessor as well. The functional firmware requirements are as follows:

- Capability to forward data from the target device to the PC.
- Ability to forward data from the PC to the target device.
- Capability to receive commands from the computer.

We use interrupts to copy the input from both UART interfaces into two circular buffers. The interrupts are used to avoid missing data transmissions while the processor is executing other code. The main program routine then performs the necessary operations on the received data. Sending data is also handled via interrupts, and there are two circular buffers for sending data as well.

The commands can vary in length and content. Some of them might set glitch parameters, and others might have data that need to be forwarded to the target. Every command starts with a one-byte header, which contains a command identifier. The command identifier determines how many bytes of data are there to read, what the data represent, and what action should be performed on the received data.

No response is sent back from the FPGA after receiving the data or executing a command unless an error occurs. The data received from the target device are sent immediately, as well as any error messages.

2.3 Software design

Another part of the fault injection platform is the software, which runs on a computer. The software offers an easy-to-use programming interface, and servers as an abstraction from data encoding. The software requirements include the following:

- Capability to set control the glitch parameters.
- Capability to receive data from the target.

We designed the software to be a counterpart of the firmware. The software includes control over the glitch parameters and the configuration of both UART interfaces. The software converts function calls into commands that are then encoded for transfer and sent over the USB-UART connection.

Chapter 3

Implementation

In this chapter, we implement the designed hardware, firmware, and software. We also write about the technologies we used during the implementation.

3.1 Hardware implementation

The hardware description is written in the Verilog hardware description language. We chose Verilog because of our prior experience with it and because it is supported by Vivado [55] design software for FPGAs, which we used to implement the hardware.

3.1.1 Hardware overview

The majority of the hardware blocks at the highest level of the design are connected via AXI4-Lite, which is a subset of AXI4. In this work, we will refer to the AXI4-Lite as the AXI for brevity.

The hardware consists of a MicroBlaze processor, glitch peripheral, GPIO AXI peripheral, two UART interfaces, Clocking Wizard, and other components that are required by the ones mentioned above, such as memory, AXI interrupt controller, and AXI interconnect.

We use a 32-bit MicroBlaze processor. MicroBlaze [56] is a RISC (Reduced Instruction Set Computer) processor, is synthesizable on an FPGA, and supports AXI. We chose MicroBlaze because it is supported by Vivado design tool, and it supports AXI, which can be used to connect it to other components. The processor has 64 KB of memory at its disposal. Originally it was 32 KB large, but the complete firmware couldn't fit in it.

A Clocking Wizard block generates a 100 MHz clock that drives every block in the design. The frequency of this generated clock also determines the glitch resolution.

There are two UART 16650 interfaces with configurable baud rates. One of them is used for communication with a computer, and the other for communication with a target device. The design also contains one GPIO block that controls a programming signal from the FPGA to the target. All three of these components are connected to the Microblaze via an AXI interconnect.

Last, a glitch IP (Intellectual Property)[1] is connected to MicroBlaze via an AXI. The peripheral is described in detail in the following section.

The top level diagram of the hardware is shown in figure 3.1. The glitch IP is highlighted in green.

[1]Intellectual property is a term used for reusable units.

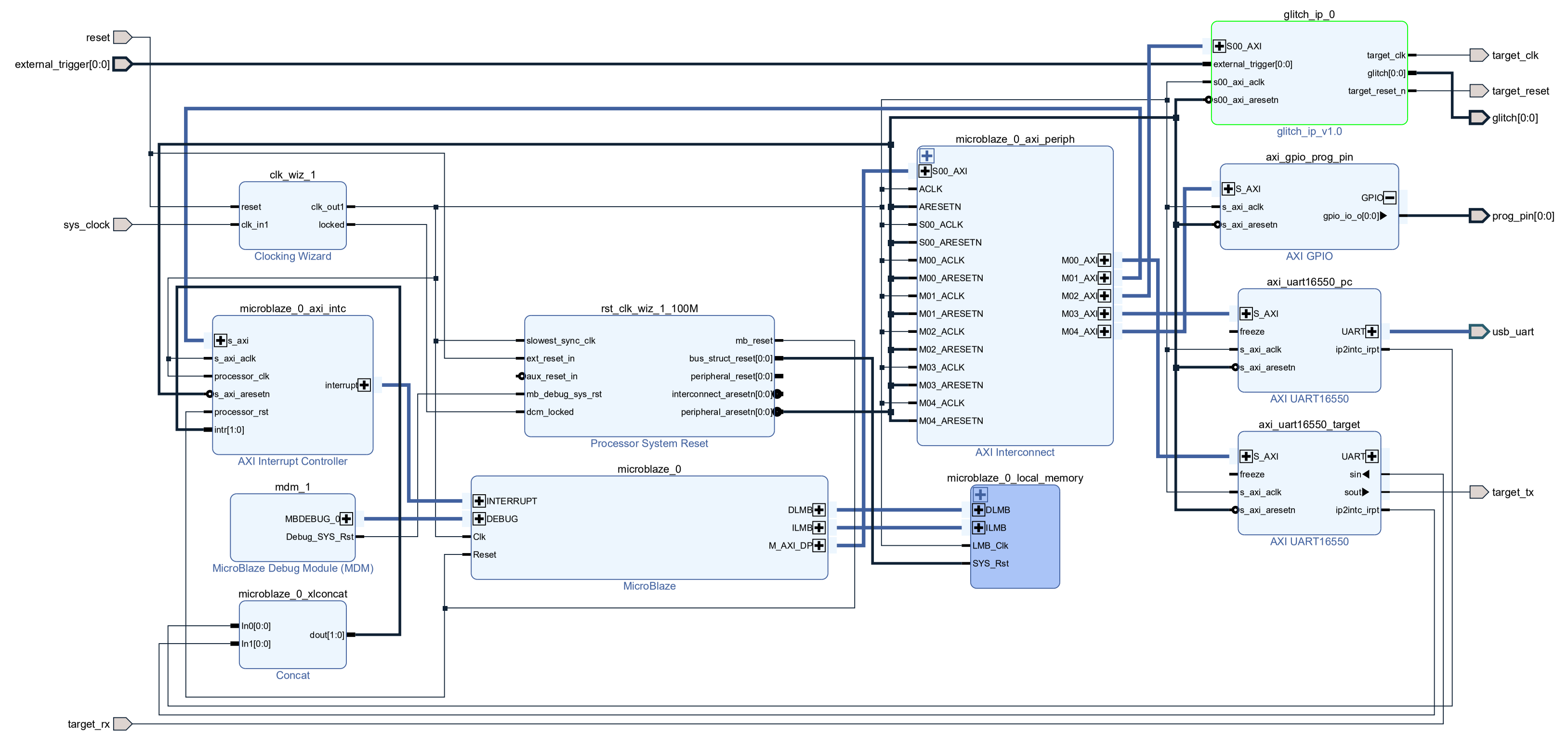


Figure 3.1 Top level diagram of the FPGA's hardware

3.1.2 Glitch IP

The glitch IP is a standalone unit that can be connected to existing designs via AXI. This peripheral is responsible for glitch insertion, target clock generation, and trigger event processing. It is divided into three modules:

- AXI module that also contains configuration registers.
- Glitch module that is responsible for glitching.
- FIFO adapter module that removes the need for the glitch module to know the exact addresses of glitch durations and glitch types.

The glitch parameters are transferred over the AXI to the configuration registers, and the values from the status registers are sent back. In addition to the AXI, the glitch IP has other inputs and outputs:

- *External* trigger input signal.
- *Target clock* output signal.
- *Glitch* output signal that controls the crowbar circuit.
- Inverted target reset output signal.

The modules of the glitch IP are parameterized to provide a good foundation for possible future modifications of the glitch module. The maximum glitch duration, the count of glitch signals, the number of trigger inputs, and the maximum glitch count are all configurable using parameters of the Verilog module.

3.1.3 AXI module

The AXI logic of this module was generated with a Vivado IP creator. Then, we modified the AXI module to output glitch parameters to the glitch module and to the FIFO adapter. The glitch parameters are the following:

- *Offset.*
- *Flags.*
- *Divisor.*
- *Glitch count.*
- *Glitch array data* that contain *glitch duration* and *glitch type* pairs.

We also added a three port memory to the configuration registers for storing thousands of *glitch duration* and *glitch type* pairs, which are used to insert multiple glitches. The three port memory was implemented to use FPGA's memory resources efficiently.

3.1.4 Three port memory

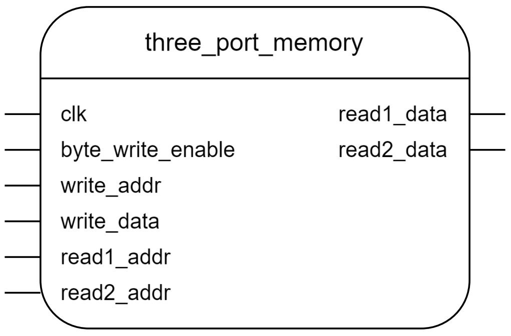
We implemented the three port memory so that it might be instantiated as BRAM, or in other words BRAM can be inferred from the code. An option to infer BRAM from the code is better than forced BRAM because the synthesis tools can choose the optimal way to generate it in the hardware. Also, the code is platform independent since it does not contain anything platform specific.

However, BRAM has some limitations. BRAM on the FPGA of CMod S7 supports up to three ports, but we needed three ports. We solved this by combining two memories together. Since we needed only one write port and two read ports, we write the same data to both memories and use the second port of every memory for reading.

BRAM inferring did not work, when we added byte write enables that are required by the AXI protocol. We had to simulate a byte write enable by reading the current value from one of the memories before writing. Another problem that we had to deal with was a mismatch between data sizes that are transferred over AXI and data sizes that are read by the glitch module through the FIFO adapter. We solved it by changing one of the memories to an asymmetric type. That means that the widths of the ports are different.

The inputs and outputs of the three port memory are shown in figure 3.2.

Figure 3.2 Three port memory I/O



3.1.4.1 True dual port memory

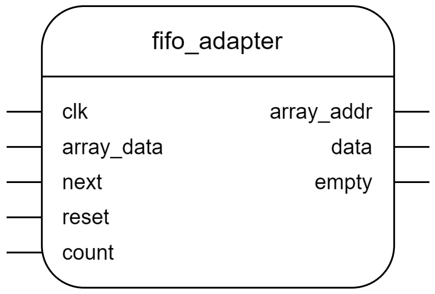
True dual port memory module is part of the three port memory and contains Verilog code for inferring true dual port BRAM. The memory is called true dual port because it contains two independent ports that can be used to read data from the memory and to write data to the memory.

One port of true dual port memory is used for getting the data requested via AXI and the other port is used for simulating byte write enable for this memory and the asymmetric memory since they contain the same data, and additionally, asymmetric memory does not support two read ports.

3.1.4.2 Asymmetric memory

Asymmetric memory is the second submodule of the three port memory. It contains a code for inferring asymmetric BRAM. It has one write port and one read port, which might be of a different width than the write port. The different read port widths are needed because the *glitch duration* and *glitch type* pair may not have the same width as the width of the data on the AXI bus, which are 32 bits wide.

Figure 3.3 FIFO adapter IO



3.1.5 FIFO adapter

We opted for inserting an adapter between the three port memory and the glitch module to abstract the location of data in the memory from the glitch module. The implementation of the FIFO adapter does not add any additional delay to the path between the memory and the glitch module because an increase in the read latency of the glitch parameters would not allow the MGIC (Multiple Glitch Insertion Controller), which is part of the glitch module, to function properly.

The input and output signals of these modules are shown in figure 3.3.When the *next* is set high, FIFO retrieves new data from the asymmetric port of the three port memory, increments the internally stored memory address, and sends the fetched data. When no new data are available for reading, the FIFO asserts the *empty* signal. The availability of data is recognized by comparing an address with the *glitch count* value. The *glitch count* indicates how many records should be read from the memory. Therefore, it also determines the number of inserted glitches. Lastly, the *reset* signal sets the internally stored address to zero.

3.1.6 Glitch module

The glitch module is a wrapper for a group of modules that together perform all operations related to glitching, including trigger signal processing and clock generation. The connection of the modules is shown in figure 3.4.

Here we list the most important modules contained in the glitch module:

- Target clock generator — Generates the *target clock* signal.
- Glitch trigger module — Controls, when the glitch insertion starts.
- Offset countdown module — Delays the glitch insertion by a fixed number of the target clock cycles.
- Multiple glitch insertion controller (MGIC) — Controls durations and types of the injected glitches.

3.1.6.1 Flags splitting module

The flags splitting module splits data from a flags configuration register into individual signals. Therefore, this module defines what bits of the flags mean. The meaning of the individual bits is shown in table 3.1.

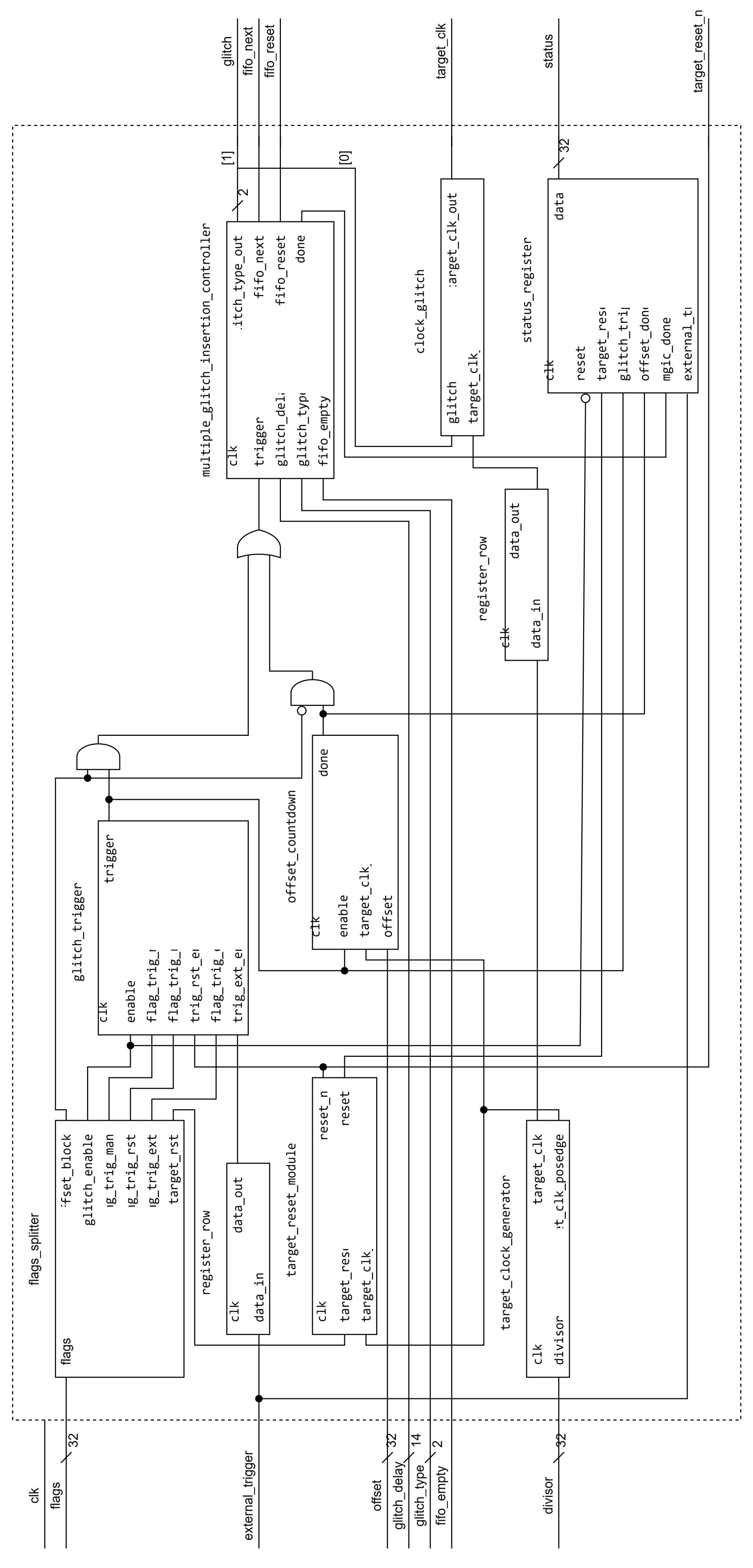


Figure 3.4 Glitch module

Table 3.1 Meaning of flags bits

Bit	Flag	Meaning
0	skip offset	When set to high, glitch insertion skips offset countdown.
2	manual trigger	Triggers glitch immediately.
3	reset trigger	Starts glitching after reset is set to low.
4	external trigger	Allows an external signal to trigger glitching.
30	target reset	Controls the output of the reset signal to the target.
31	glitch enable	Enables glitch injection. It must be set low after an insertion.

3.1.6.2 Target clock generator module

This module generates a *target clock* signal by dividing the main 100 MHz clock that is generated by the Clocking Wizard. *Divisor* input determines how many clock cycles of the main clock a half-period of the target clock lasts. This module also generates a signal one clock cycle before the rising edge of the target clock to allow other modules to synchronize the glitch insertion with the *target clock*.

The inputs and outputs are shown in figure 3.5, and the clock generator state diagram is shown in figure 3.6.

Figure 3.5 Target clock generator I/O

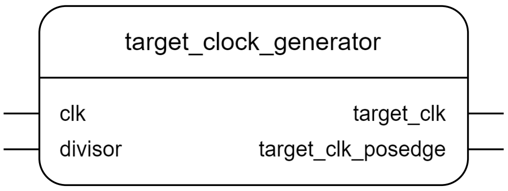
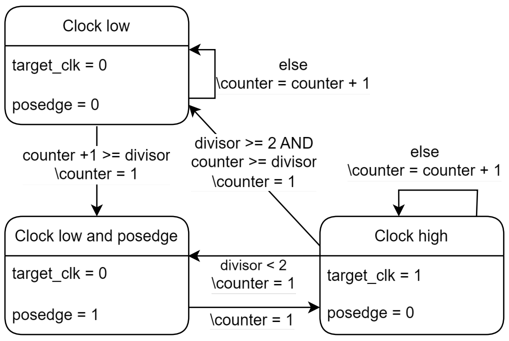


Figure 3.6 Target clock generator state machine



3.1.6.3 Target reset module

The value of *target reset* is controlled by a single bit from the flags register, which is denoted as the target reset flag. When the reset flag transitions to low, the reset signal does too, but the transition of the reset signal from high to low only occurs on the rising edges of the target clock to ensure that the trigger signal, to which the *target reset* is connected, always occurs at the same time relative to the target clock.

3.1.6.4 Glitch trigger module

The glitch trigger component evaluates the state of the flags register and trigger signals, and outputs a single *trigger* signal enables offset countdown module or MGIC, if the offset countdown is skipped. The *trigger* is set high when an input *trigger* signal is set high, a corresponding flag is asserted, and the glitching is enabled. Only one of the trigger signals with its enable signal has to be set high to assert the *trigger* signal.

3.1.6.5 Offset countdown module

The offset countdown module delays the glitch insertion by a number of target clock cycles, which is stored in a glitch offset configuration register. The countdown starts when the *trigger* signal from the glitch trigger module is asserted. This module can be skipped during a glitch insertion by setting a flag in a configuration register. After the countdown has finished, the *offset countdown done* signal is set high.

The module's I/O and its state diagram are shown in figures 3.7 and 3.8 respectively.

Figure 3.7 Offset countdown I/O

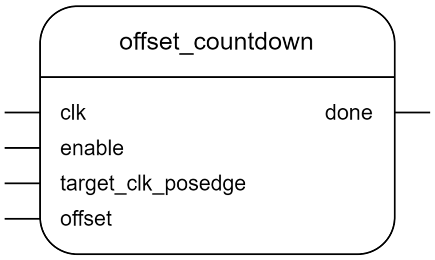
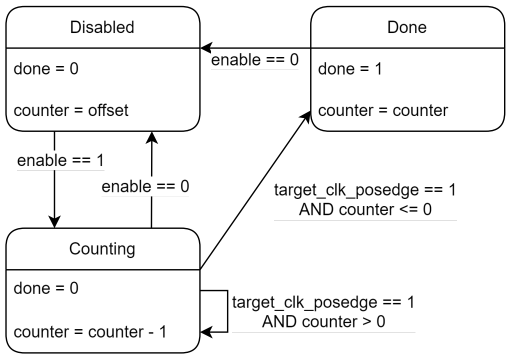


Figure 3.8 Offset countdown state diagram



3.1.6.6 Multiple glitch insertion controller

Multiple glitch insertion controller manages when glitches are inserted and what types of glitches are injected. This module does not handle how glitches are inserted, but it generates enable signals for the modules that do the actual glitching.

The glitching starts if one of two conditions is satisfied. Either the *offset countdown done* is set high, or a trigger event is received, and the skip offset is asserted via its flag bit.

First, we describe the general principle behind the MGIC. The controller receives *glitch duration* and *glitch type* pairs from the FIFO adapter, and outputs the received *glitch type* for the number of clock cycles specified by the *glitch duration*. After the duration has passed, another *glitch type* is output for a new *glitch duration* that is again received from the FIFO. This continues until the FIFO asserts the *empty* signal. After that, a *MGIC done* signal is set high to signal that the glitching is completed.

Now we write about the MGIC implementation in greater detail. This module supports glitch durations of one clock cycle and longer. The upper limit is given by the width of the glitch duration input bus. Since the memory behind the FIFO adapter has a one clock-cycle read latency, we fetch the glitch parameter pairs before the previous glitch duration passes.

When the glitching starts, the second *glitch type* and *glitch duration* pair is requested from the FIFO by raising the *next* signal. The second pair is requested before a glitch is inserted because there would not be enough time to insert the second glitch on time if the first *glitch duration* was a single clock cycle.

After the second glitch pair is requested, a first glitch is inserted for the number of FPGA's clock cycles specified by the *glitch duration*. In the last cycle of a glitch, another pair of parameters is requested. This continues until the FIFO adapter asserts *empty* signal, which means that all glitches were inserted. After that, all glitch signals are cleared and a *MGIC done* signal is set high, and written into a status register in a separate module.

The MGIC introduces a two-cycle delay on the *glitch type* bus. Due to this, the *target clock* signal has to be also delayed by two clock cycles to preserve their relative offset. The *target clock* is delayed even when the glitching is not active to keep the period of the target clock stable.

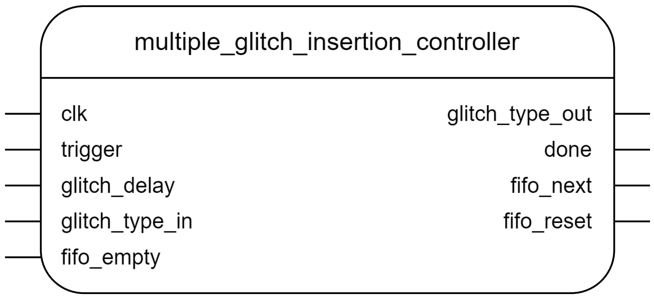


Figure 3.9 Multiple glitch insertion controller I/O

To better vizualize the function of this module, we provide an I/O diagram in figure 3.9, and the state diagram in figure 3.10

3.1.6.7 Register row module

We used this module to introduce signal delay for synchronization purposes of a *target clock* and *glitch* signals and to prevent clock domain crossing issues when reading trigger signals from the target.

3.1.6.8 Clock glitch module

The clock glitch module performs the XOR operation of a *target clock* signal and a *glitch* signal, which is generated by the MGIC. The clock glitch is controlled by the lowest bit of the *glitch* signal.

3.1.6.9 Status register module

The status register gives feedback on the progress of the glitch insertion process to the user. It tracks the following events and signal values:

- External trigger values.
- Glitch triggered event.
- End of the offset countdown.
- End of the whole glitch insertion process, i.e. the moment after all glitches are inserted.
- *Target reset* signal value.

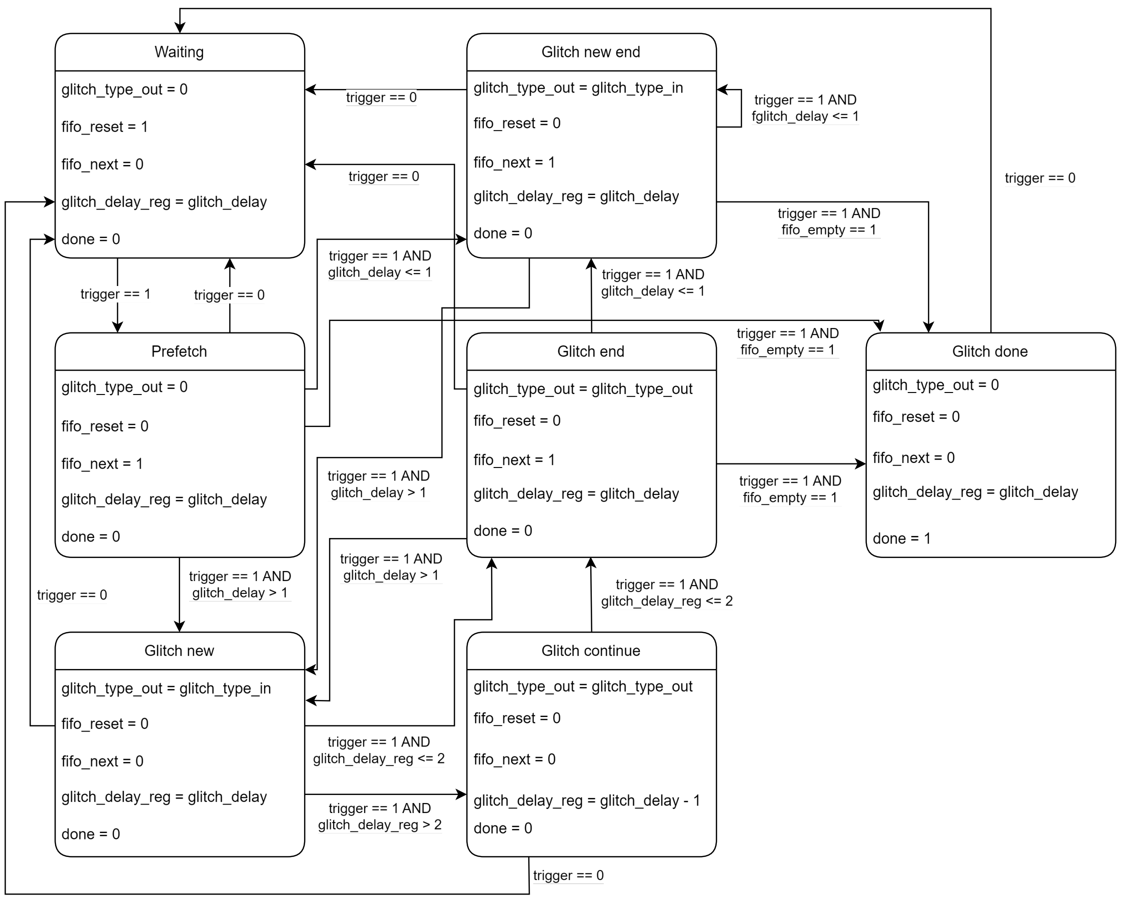
The *external trigger* values are cleared after the glitch insertion is disabled because otherwise short signals could easily be missed when reading the value of the status register.

3.2 Firmware implementation

The role of the firmware is to provide an interface through which the platform can be configured. The firmware receives commands from a computer, and then it either configures the glitch insertion parameters or interacts with the target.

We wrote the firmware in C programming language because it is supported by the Vitis [57] integrated development environment. We used Vitis for firmware development and for FPGA programming.

Figure 3.10 Multiple glitch insertion controller state diagram


Waiting
glitch_type_out = 0
fifo_reset = 1
fifo_next = 0
glitch_delay_reg = glitch_delay
done = 0
Glitch new end
glitch_type_out = glitch_type_in
fifo_reset = 0
fifo_next = 1
glitch_delay_reg = glitch_delay
done = 0
Prefetch
glitch_type_out = 0
fifo_reset = 0
fifo_next = 1
glitch_delay_reg = glitch_delay
done = 0
Glitch end
glitch_type_out = glitch_type_out
fifo_reset = 0
fifo_next = 1
glitch_delay_reg = glitch_delay
done = 0
Glitch done
glitch_type_out = 0
fifo_reset = 0
fifo_next = 0
glitch_delay_reg = glitch_delay
done = 1
Glitch new
glitch_type_out = glitch_type_in
fifo_reset = 0
fifo_next = 0
glitch_delay_reg = glitch_delay
done = 0
Glitch continue
glitch_type_out = glitch_type_out
fifo_reset = 0
fifo_next = 0
glitch_delay_reg = glitch_delay - 1
done = 0
trigger == 0
trigger == 0
trigger == 1 AND fglitch_delay <= 1
trigger == 0
trigger == 1 AND fifo_empty == 1
trigger == 1 AND glitch_delay <= 1
trigger == 1 AND glitch_delay <= 1
trigger == 1
trigger == 0
trigger == 1 AND fifo_empty == 1
trigger == 1 AND glitch_delay > 1
trigger == 1 AND fifo_empty == 1
trigger == 1 AND glitch_delay > 1
trigger == 0
trigger == 1 AND glitch_delay > 1
trigger == 1 AND glitch_delay_reg <= 2
trigger == 1 AND glitch_delay_reg <= 2
trigger == 1 AND glitch_delay_reg > 2
trigger == 0

3.2.1 Commands

Communication between the computer and the FPGA is command-based. The PC sends a command, and the FPGA performs an operation defined by the command and optionally sends some data back. Commands are identified by the first byte of a transfer. The length of the command's content depends on the command. The available commands are:

- Forward data that forwards the received data to the target.
- Set offset that writes the value to the glitch offset register in the glitch IP.
- Set flags command that sets glitch flags.
- Set divisor command sets the ratio between the target clock and the FPGA's 100 MHz clock. The sent value defines how many clock cycles a target's clock half-period lasts.
- Set glitch count that determines how many elements are valid in the glitch array.
- Set glitch array that overwrites the glitch type and the glitch duration pairs from a specified offset.
- Get glitch status that returns the content of the status register in the glitch IP.
- Get info command returns the platform's hardware capabilities such as maximum glitch count.
- Read register reads any register in the glitch IP.
- Write register writes to any register in the glitch IP.
- Enter programming mode command starts infinite forwarding of data between the PC and the target without the need for the send data command, and asserts the *programming* signal.
- Set PC UART baud rate changes the baud rate of the UART that is connected to the computer.
- Set target UART baud rate changes the baud rate of the target facing UART.

The data that are received from the target, are immediately forwarded to the PC.

Upon receiving an invalid message, the program execution stops in order not to cause any harm to the target by executing invalid commands. The platform has to be reset by pressing *BTN0* on Cmod S7.

3.2.2 AXI glitch

Constants such as glitch type width, glitch delay width, and maximum glitch count are defined as macros. When the hardware configuration is changed, these values have to be manually updated to correspond with the hardware's capabilities.

3.2.3 Circular buffer

We implemented a circular buffer to store data received over UART in an interrupt service routine. A circular buffer is a fixed-size structure that maintains a pointer to the start and the end of the stored data. Retrieving data moves the pointer pointing at the start of the data and writing moves the second pointer.

Data from our implementation of the circular buffer are obtained by requesting a pointer to the start of the data. After the required operation is performed on the data, the data are removed from the buffer by calling a function that moves the start of the buffer. The two-step process is used to avoid creating unnecessary copies of the data that could be used directly from the buffer.

3.2.4 UART

Both UART interfaces are configured to eight bits of data length with one stop bit and without parity. Also, they feature 16 character transmit and receive FIFOs. The default baud rate of both UART interfaces is 9600 bauds. The baud rate can be changed by sending a designated command to the FPGA.

The two UART interfaces are interrupt-driven. We set up interrupt service routines to store the received data in circular buffers. Each UART has its own buffer. The UARTs can transmit data from the circular buffers, or a different buffer can be specified.

3.3 Software implementation

We created an interface for communication with the FPGA in the Python programming language. We implemented the interface in the form of a Python class that stores the hardware configuration and provides abstraction from the communication protocol between the computer and the FPGA. The software includes functions that send all of the commands mentioned in the firmware implementation section 3.2.1. In addition, there is a function for reading data from the UART interface, and there are also some other functions that improve the code readability and user experience.

We implemented the software in such a way that it does not abstract from the inner workings of the fault injection platform. Listing 3.1 shows an example of how to insert a clock glitch with the implemented interface. In the example, the clock glitch is inserted with offset of one and duration of two and is triggered by an external trigger.

Code listing 3.1 Clock glitch insertion example using our platform

```
dev.set_offset(1)
dev.set_glitch_count(1)
dev.set_glitch_array([(dev.GLITCH_CLOCK, 2)], 0)
dev.set_flags(dev.FLAG_GLITCH_ENABLE |
    dev.FLAG_FIRST_EXTERNAL_TRIGGER)
```

We used the pySerial module to handle communication over the serial line. pySerial [58] is a Python module for accessing serial ports. The software also provides the option to log the communication and the option to send all data in one burst.

We also created a file with examples that show how to use the platform. The examples are also in Python. The examples include clock glitching, voltage glitching, various trigger possibilities, and multiple fault injection. The examples are implemented in file `Examples.ipynb`.

Chapter 4

Hardware testing

Due to the complexity of the hardware, we created simulation testbenches for every module in the glitch IP. We wrote the testbenches in Verilog. The testbenches cover all of the glitch peripheral's Verilog modules. It was important to catch hardware design and implementation errors before working on the software because it decreased the time spent debugging the code.

We also checked the glitch output of the implemented fault injection platform with an oscilloscope. Figure 4.1 shows three clock and voltage glitches with widths of 10 ns, 50 ns, and 100 ns. The delays between glitches were 10 ns, and 50 ns. Figure 4.2 shows 10 clock glitches and 10 voltages glitches with duration of 10 ns and delays of 10 ns synchronized with *target clock* rising edge. Lastly, figure 4.3 shows also 10 glitches but with a greater magnification.

In all three figures, channel one is *target clock*, channel two is *trigger*, and channel three is *glitch* signal, which is connected to the MOSFET of the crowbar circuit. Channel four is unimportant. All shown glitches were triggered by the *trigger* signal from a target.

Figure 4.1 10 ns, 50 ns and 100 ns clock and voltage glitches inserted using our platform

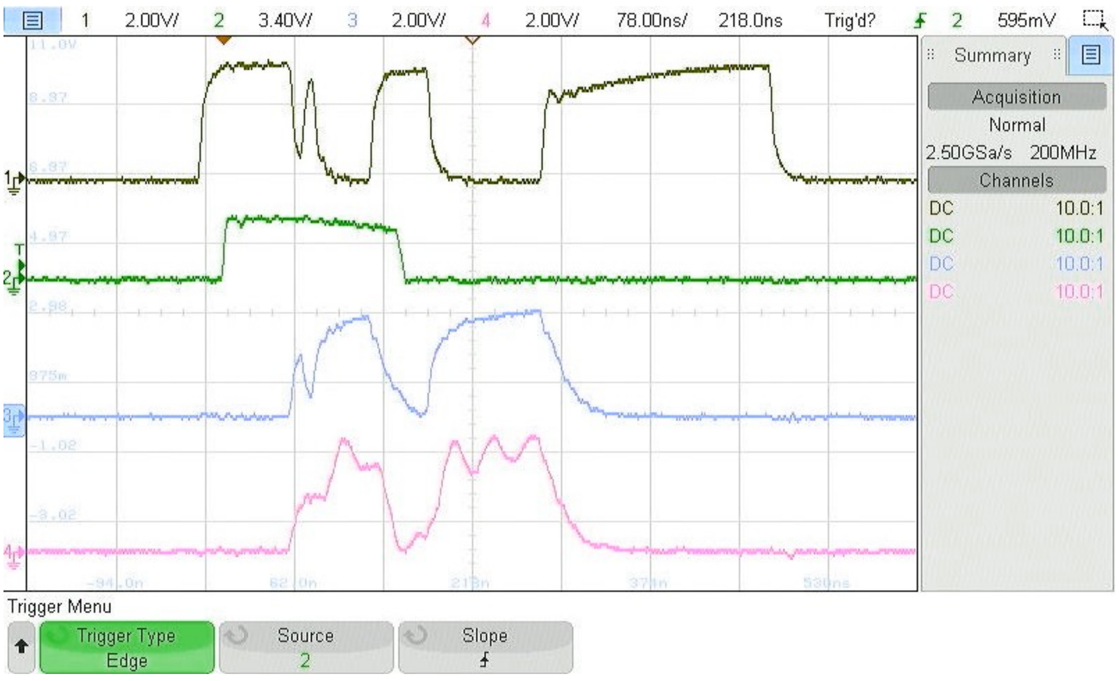


Figure 4.2 10 10 ns clock and voltage glitches inserted using our platform

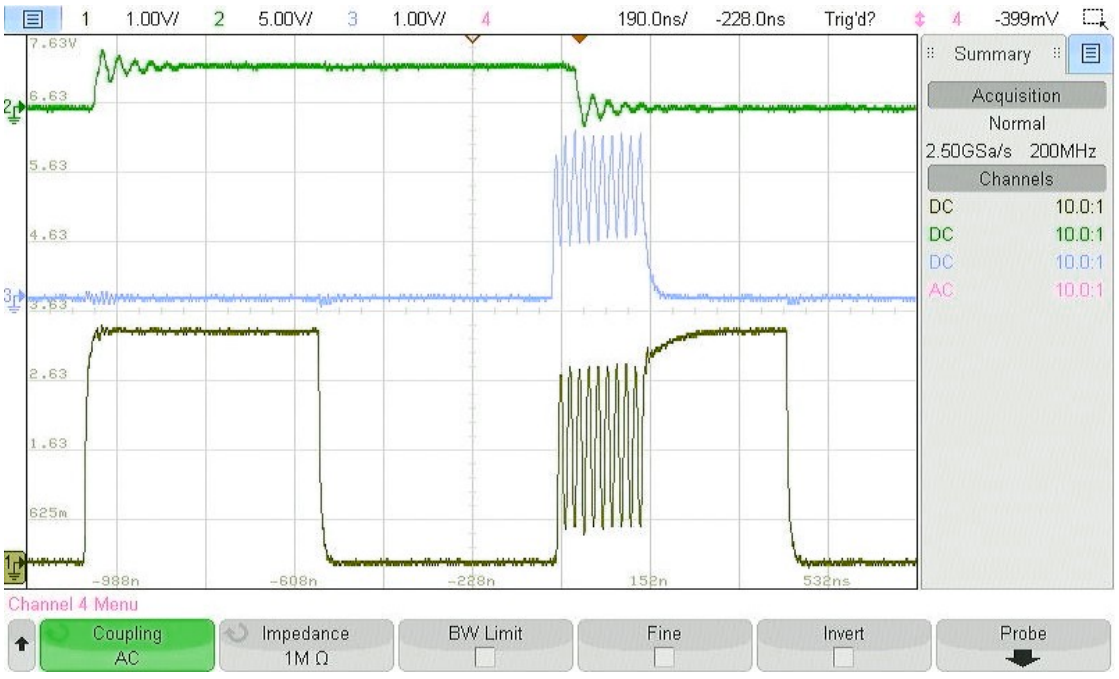
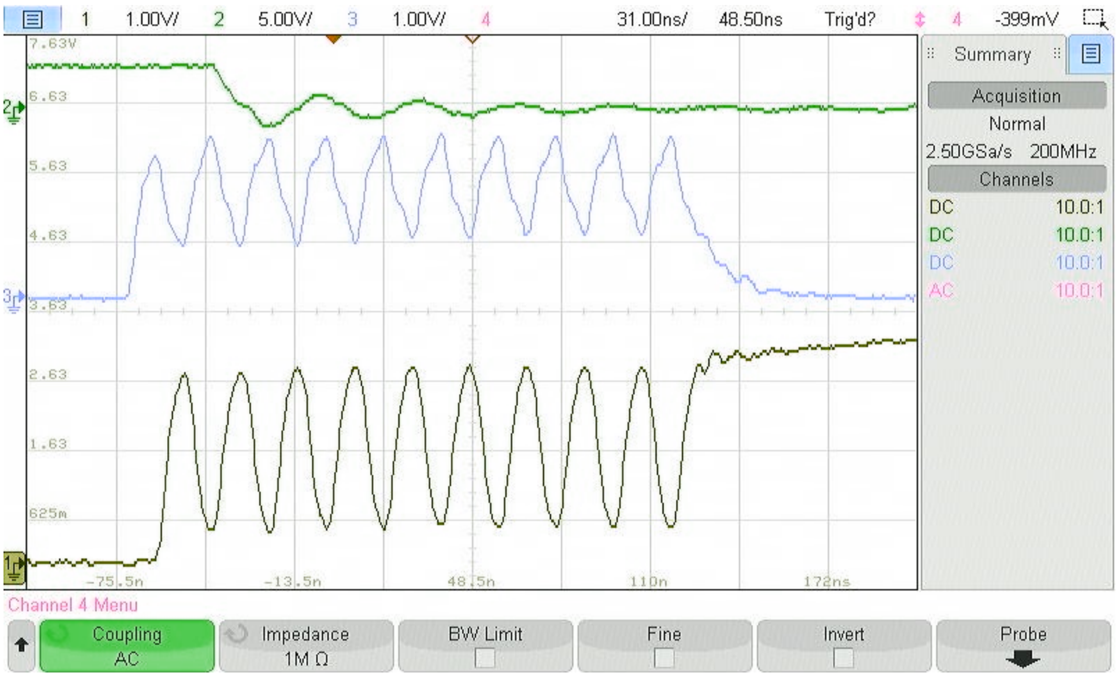


Figure 4.3 Closeup of 10 10 ns clock and voltage glitches inserted using our platform



Chapter 5
Fault injection attacks implementation and results

First, we describe the target device. Then, we write about the attacks that we carried out on the AES cipher. We performed two attacks on the AES using our fault injection platform and one using the ChipWhisperer-Nano for comparison.

In this chapter, we use the term glitch delay to refer to a delay in FPGA's clock cycles before a glitch is inserted, and the term glitch width to refer to a duration of the glitch. However, both are implemented as a glitch duration in the hardware. We write about the glitch duration in subsection 2.1.4.

5.1 Target device

An ATmega8L was chosen as the target of the attacks carried out with our fault injection platform. It was chosen because the AES implementation for the microcontroller was provided by the supervisor. ATmega8L [59] is an 8-bit microcontroller built on top of an AVR RISC architecture.

ATmega8L microcontroller has some features that we used to carry out the attacks. We used a feature of ATmega8L to run off an external clock signal to synchronize glitches with the target's clock. Also, we used the ATmega8L's external reset as another type of synchronization between the target and the fault injection device.

Brown-out detection of the microcontroller was turned off prior to carrying out the attacks. The target clock frequency of the microcontroller was 1 MHz. The target clock signal was generated by our fault injection platform.

We used the same plaintext and ciphertext for all attacks using our platform. The plaintext encrypted by the AES was in the hexadecimal notation 50DBE8E2 4B20B170 8E1EFBE7 B31FDADB. 123456AB 254600FF DEADBEEF 00CAFE34 was the key that we used to encrypt the plaintext.

5.2 Wiring of our platform

The wiring consisted of the Cmod S7 development board, a perfboard with ATmega8L, a crowbar circuit, and a PMOD header. The perfboard was provided by the supervisor. However, we added connections for *programming* and *external trigger* signals.

We routed the I/O pins of the hardware implemented on Cmod S7 to its PMOD header. The I/O pins were assigned as follows:

- J2 (Pin 1) — Sout, data to the microcontroller.
- H2 (Pin 2) — Sin, data from the microcontoller.
- H4 (Pin 3) — Reset signal.
- F3 (Pin 4) — Target clock signal.
- Pin 5 — GND.
- Pin 6 — VCC.
- H3 (Pin 7) — Trigger signal.
- H1 (Pin 8) — Programming signal.
- G1 (Pin 9) — Unassigned.
- F4 (Pin 10) — Glitch signal.
- Pin 11 — GND.
- Pin 12 — VCC.

The PMOD header of Cmod S7 was connected to a PMOD header on the perfboard. A circuit diagram of the perfboard is shown in figure 5.1. Q1 denotes an N-channel BS170 MOSFET, and RV1 is a 200 Ω multi-turn trimmer potentiometer, or to be more exact, M64Y201KB40 made by Vishay. We set the resistance R_{AB} of the trimmer RV1 to 150 Ω, and the resistance R_{BC} to 50 Ω. We started with these resistances, and we found that voltage glitching was successful with them. We did not experiment with adjusting the resistances.

Figure 5.2 shows Cmod S7 that is connected to the perfboard with ATmega8L, which is the 28-pin package in the figure. MOSFET, trimmer, and some pins for observing signals with an oscilloscope are also visible.

5.3 Attack on AES between the 8th and the 9th *MixColumns* using our platform

This section contains the implementation of the attack and the results of the attack on AES by injecting a fault between the 8th and the 9th *MixColumns*.

5.3.1 Implementation

This attack requires multiple pairs of ciphertexts and erroneous ciphertexts. Erroneous ciphertexts are obtained by injecting a fault between the 8th *MixColumns* and the 9th *MixColumns*.

To insert glitches at a specific point in the AES execution, we reset the target device before every fault injection. Then, we injected the glitch after a certain number of target clock cycles after setting the *target reset* signal low.

Firstly, we had to determine when to insert glitches. We obtained the approximate offsets from the power trace of the ATmega8L by inserting glitches at various offsets and observing the power trace. The offset specified after how many target clock cycles from lowering the *target reset* signal a glitch was inserted. The power trace is shown in channel four in figure 5.3. We assumed that the 10 peaks in figure 5.3 corresponded to 10 AES rounds and chose offsets based on that

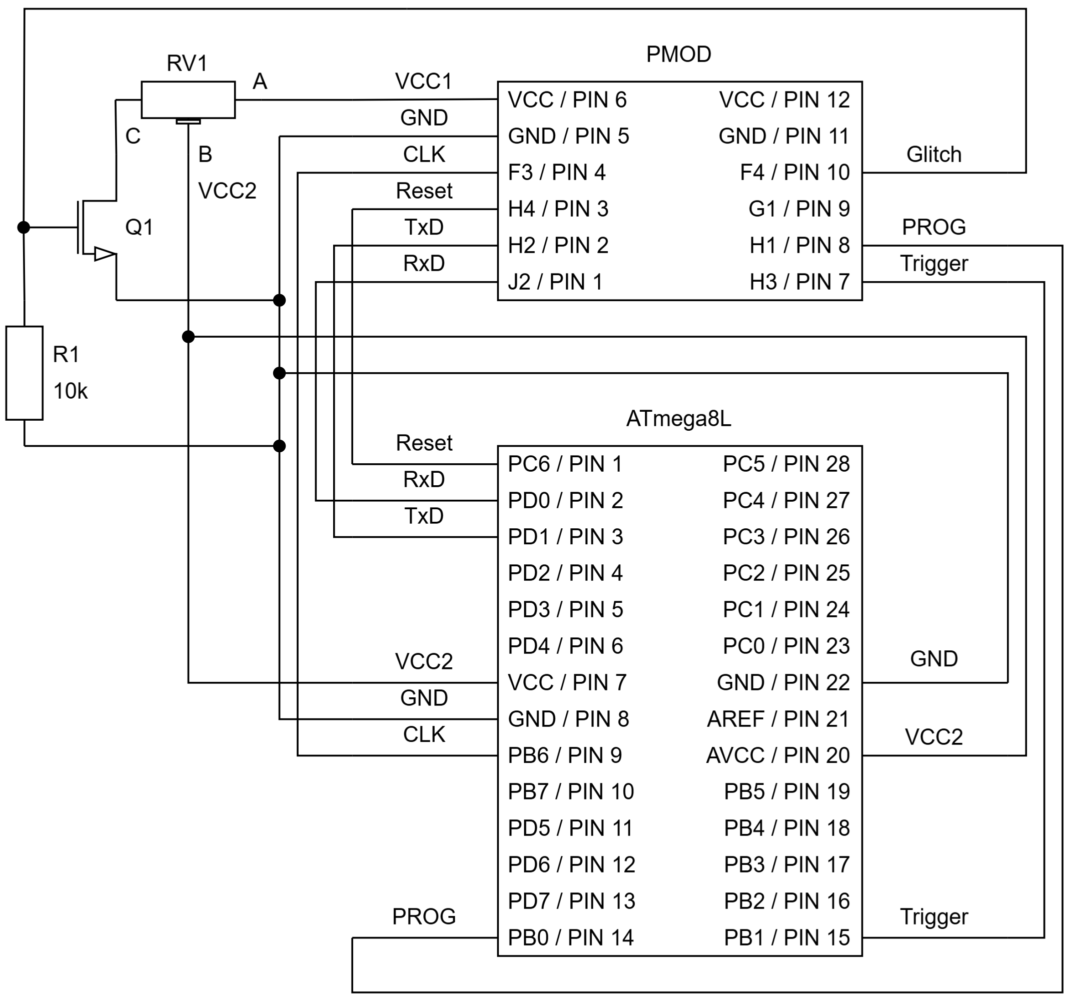


Figure 5.1 Circuit diagram of the perfboard with ATmega8L

assumption. Further investigation was not necessary because this attack does not require precise timing, and we confirmed the correctness of our assumption by the characteristic placement of the four faults in the faulty ciphertexts and successfully carrying out the attack.

We tried voltage glitching and clock glitching to induce faults between the 8th and the 9th *MixColumns* of the AES. In both cases, we injected faults at offset in range 12700–13800 since this range corresponded to the 8th and 9th peak in power consumption in the power trace mentioned above. While voltage glitching, we tried different widths of the glitch in increments of 1 µs, but we kept the glitch delay at zero. We inserted clock glitches with the shortest possible width of 10 ns. We tried locations near target clock edges by adjusting the delay of the glitch. Since the target clock frequency was 1 MHz. Glitches with delays from 0 to 49 were inserted when the target clock signal was set high, and the glitches with delays from 50 to 99 were injected when the target clock signal was set low. The code for injecting faults is in file `fi_cmod.ipynb`.

We implemented algorithm 1.5 in Python without modification for the attack with injected faults between the 8th and the 9th *MixColumns* to recover the last round key. The algorithm is implemented in file `dfa_89.ipynb`. Also, we implemented functions to filter out ciphertexts that did not have four faults in the required bytes. These functions are implemented in file `aes_masks`. This file also contains the definitions of the required positions of faulty bytes in the ciphertexts. Finally, we implemented the inverse key expansion algorithm in a separate file

Figure 5.2 Picture of Cmod S7 connected to the perfboard with ATmega8L

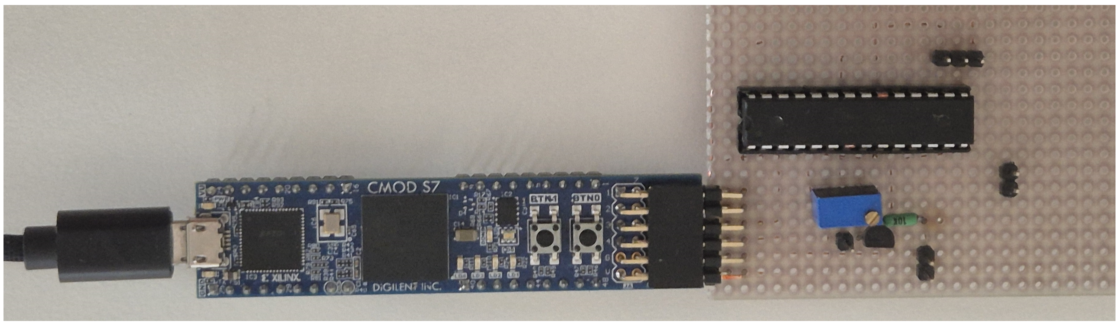
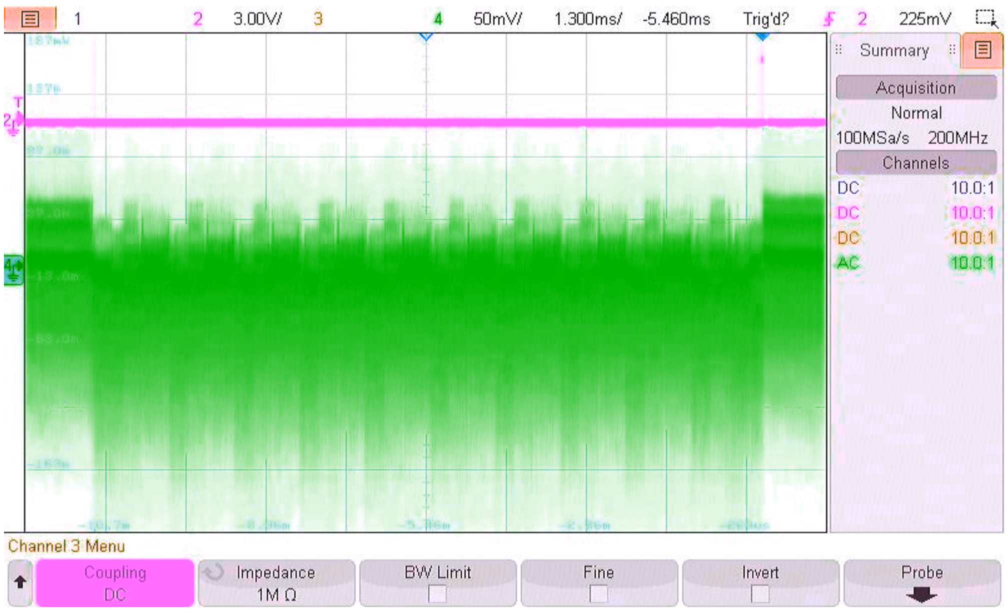


Figure 5.3 AES power trace



`aes_inverse_key_expansion.ipynb`, so it can be reused for the other attack on the AES.

Then, we performed the attack with the described glitch settings and the implemented algorithms.

5.3.2 Results

We injected 10 voltage glitches at each offset. In figure 5.4, the number of faulty ciphertexts is shown based on the offset and the affected column of the AES *state*. Each unit of offset corresponds to one target clock cycle, which was 1 µs long. Figure 5.5 shows how the number of recovered faulty ciphertexts changed with the width of the voltage glitch. One unit of width and glitch delay is equal to 10 ns. Voltage glitching was the most effective with the glitch width set to 1 µs or 2 µs.

We inserted 8 faults when using clock glitching. Figure 5.6 shows how the number of erroneous ciphertexts changes with the glitch offset. The ciphertexts were divided into groups based on the column into which a fault was inserted. The effects of the delay of the glitch on the number of induced faults are shown in figure 5.7. The best clock glitching location is right before and right after the rising edge of the target clock.

Figure 5.4 Voltage glitching faults by offset obtained by glitching with widths from 100 to 1000 in increments of 100 using our platform

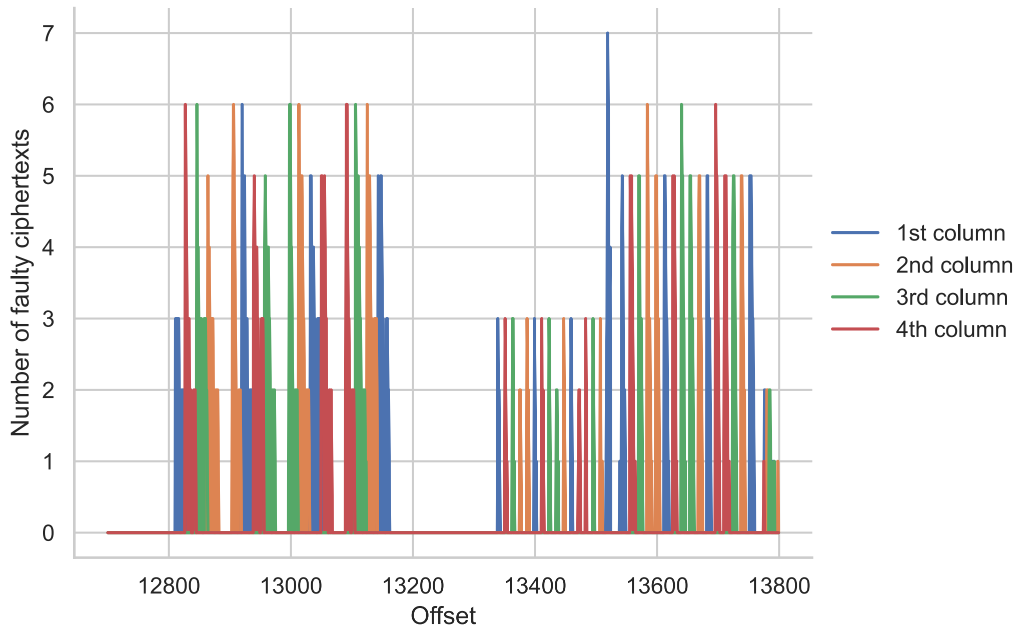
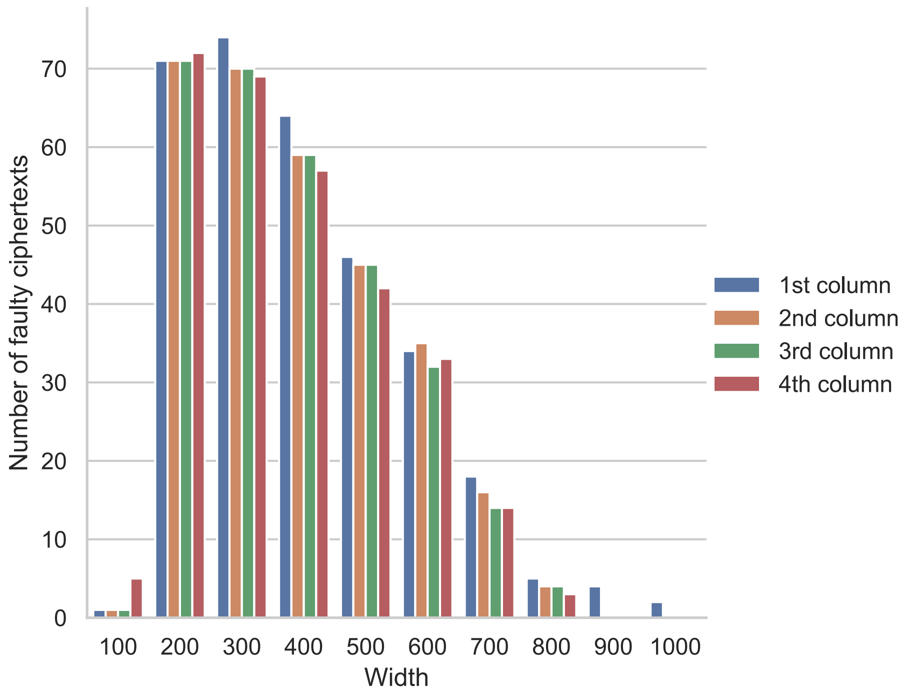
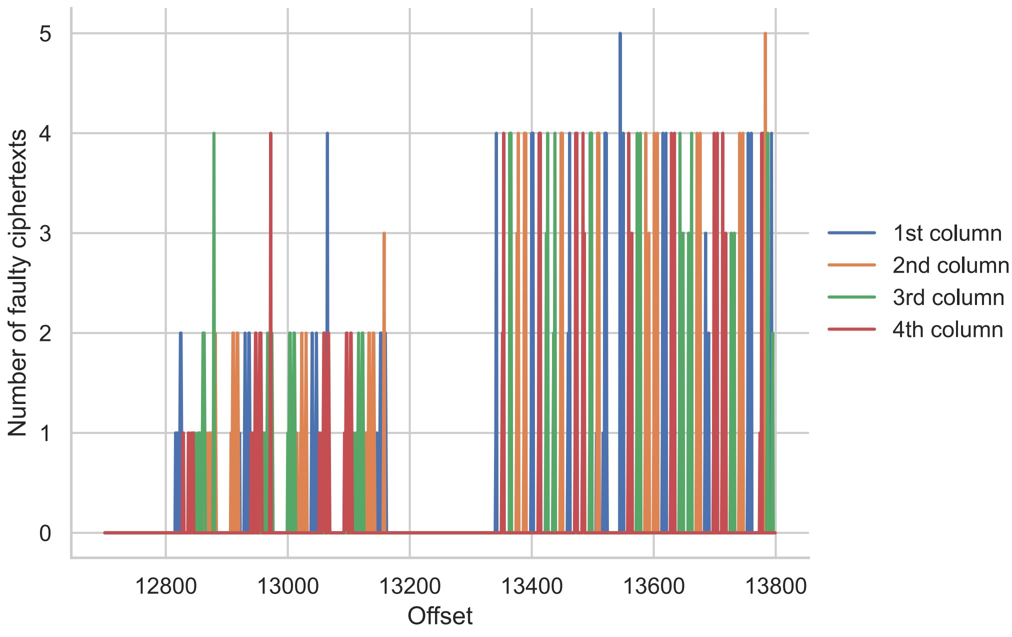


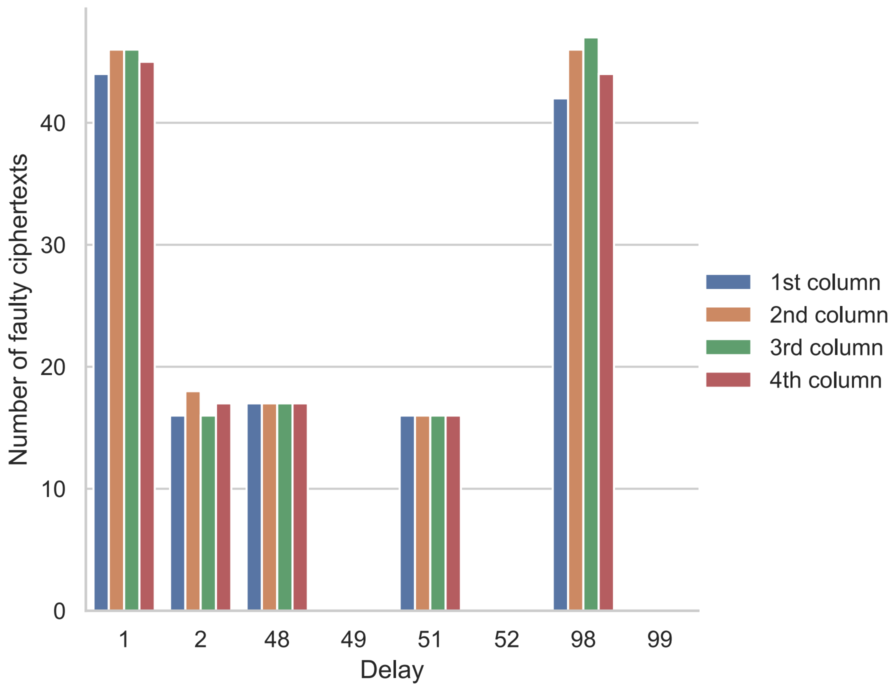
Figure 5.5 Voltage glitching faults by width obtained by glitching with offsets from 12700 to 13800 using our platform



🟦 **Figure 5.6** Clock glitching faults by offset obtained by glitching with delays of 1, 2, 48, 49, 51, 52, 98, and 99 using our platform



🟦 **Figure 5.7** Clock glitching faults by delay obtained by glitching with offsets from 12700 to 13800 using our platform



The ciphertexts obtained via voltage glitching are available in file `aes_voltage_reset_89.npy` and the data from the graph are stored in `aes_voltage_reset_89_settings.npy`. The file with the erroneous ciphertexts obtained via clock glitching is named `aes_clock_reset_89.npy`, and the file that contains the data on the success rate of clock glitching is `aes_clock_reset_89_settings.npy`.

We ran the key recovery algorithm for the ciphertext pairs obtained by voltage glitching and clock glitching. In both cases, we were able to successfully recover the AES encryption key after the inverse expansion of the last round key, which was returned by the implemented key recovery algorithm. The value of the recovered key was 123456AB 254600FF DEADBEEF 00CAFE34.

5.4 Attack on AES between the 7th and the 8th *MixColumns* using our platform

We also carried out attacks on the AES by injecting faults between the 7th and the 8th *MixColumns*. We used only voltage glitching this time.

5.4.1 Implementation

We injected voltage glitches into every 10th offset from 11600 to 13000. The glitch delay was set to zero or 50 and the widths of the glitches that we tried were 100, 200, 300, and 400 duration units of our platform, which are equal to 10 ns. The glitch was triggered by the rising edge of inverted *target reset* signal. We used our Python API to insert glitches with our platform. The code for injecting faults is in file `fi_cmod.ipynb`.

We implemented algorithms 1.5, A.1, and 1.3, which was modified for the attack between the 7th and the 8th *MixColumns*. They were implemented in Python and are located in the files `dfa_78_simple.ipynb`, `dfa_78_piret.ipynb`, and `dfa_78_dusart`, respectively. We also implemented algorithm A.2 to reduce the key space for algorithm A.1.

We used the implementation of algorithm 1.5 to find ciphertext pairs that would lead to successful key recovery. To achieve this, we tried combinations of two of the obtained plaintext and ciphertext pairs to recover the 10th round key. We search for combinations of two ciphertext pairs because some erroneous ciphertexts could have a fault that could not be used to recover the last round key.

If we get combinations of ciphertexts that led to a successful recovery from algorithm 1.5 in `dfa_78_simple.ipynb`, we would only validate the implementations of algorithms A.2 and 1.5 by executing them with the working combinations of ciphertext pairs.

The recovered last round keys had to go through an inverse key expansion, which was implemented in `aes_inverse_key_expansion.ipynb`.

5.4.2 Results

We were able to recover the encryption key 123456AB 254600FF DEADBEEF 00CAFE34 that matched the encryption key on the target. Table 5.1 contains 10 ciphertext pairs that lead to a successful key recovery using algorithm 1.5. All obtained erroneous ciphertexts are available in file `aes_voltage_reset_78_cts.npy`. We also collected data on the glitching success rate, which contain counts of faulty ciphertexts per glitch offset and glitch width. They are stored in `aes_voltage_reset_78_settings.npy`.

The implementation of algorithm A.1 with the key space reduction algorithm A.2 needed the first two pairs of ciphertexts from table 5.1 to recover the last round key. After executing inverse key expansion algorithm, the encryption key was recovered.

Finally, the implementation of algorithm 1.3 needed all 10 ciphertext pairs listed in table 5.1 to recover the last round key.

Table 5.1 Pairs of correct and erroneous ciphertexts obtained by injecting faults between the 7th and the 8th *MixColumns*

Correct ciphertexts	Erroneous ciphertexts
`EA5B2DC7 D39C2B38 170AC892 24539B1F`	`7A9FD066 4ED51722 61E72800 1FC89FE2`
`EA5B2DC7 D39C2B38 170AC892 24539B1F`	`A7254265 89D1E066 E26F84AE 9FBD2770`
`EA5B2DC7 D39C2B38 170AC892 24539B1F`	`52F85501 8B4374C5 1AAB9A34 02010748`
`EA5B2DC7 D39C2B38 170AC892 24539B1F`	`BAC9F0C6 B08D35FB D9D542EC 7177A4C0`
`EA5B2DC7 D39C2B38 170AC892 24539B1F`	`29BFFF75 B1223E8A B24411CC D451D53C`
`EA5B2DC7 D39C2B38 170AC892 24539B1F`	`42D4395B 380FC4F3 B0B260DE 1534C982`
`EA5B2DC7 D39C2B38 170AC892 24539B1F`	`BF36E1B0 0C25CE24 03F6D19E DEB16C5E`
`EA5B2DC7 D39C2B38 170AC892 24539B1F`	`B20F9734 D93D8166 2928A763 8D585C25`
`EA5B2DC7 D39C2B38 170AC892 24539B1F`	`19FDD1EC 636C19CA 92022F80 90C033CD`
`EA5B2DC7 D39C2B38 170AC892 24539B1F`	`2F78A85E B15CA2C3 CB52208C F26BBC1A`

5.5 Attack on AES between the 8th and the 9th *MixColumns* using ChipWhisperer-Nano

We also performed the attack on AES that exploits faults between the 8th and the 9th *MixColumns* with ChipWhisperer-Nano.

5.5.1 Implementation

We carried out the attack with ChipWhisperer-Nano on a different target. The target was the built-in STM32F303F4P6. The firmware version of ChipWhisperer-Nano was 0.65.0. The AES key on the target was 01020304 05060708 090A0B0C 0D0E0F10, and the plaintext was equal to 00000000 00000000 00000000 00000000.

Before the attack, we found that the glitch settings with *repeat* of 3 with combination of 30 MHz target clock frequency induced the faults the most reliably. The *repeat* is equal to glitch duration setting on our platform. We obtained the offsets, which correspond to the interval between the 8th *MixColumns* and the 9th *MixColumns*, from the power traces. They were captured using the ADC on the ChipWhisperer-Nano. The offset range used was 33000–35000.

We controlled the ChipWhisperer-Nano via its Python API. The code responsible for injecting faults is located in file `fi_89_cw_nano.ipynb`. Then we used algorithm 1.5 in file `dfa_89.ipynb` without modification for the attack between the 7th and the 8th *MixColumns* to recover the last round key. Then, we used the implemented inverse key expansion algorithm 1.4, which is implemented in file `aes_inverse_key_expansion.ipynb`.

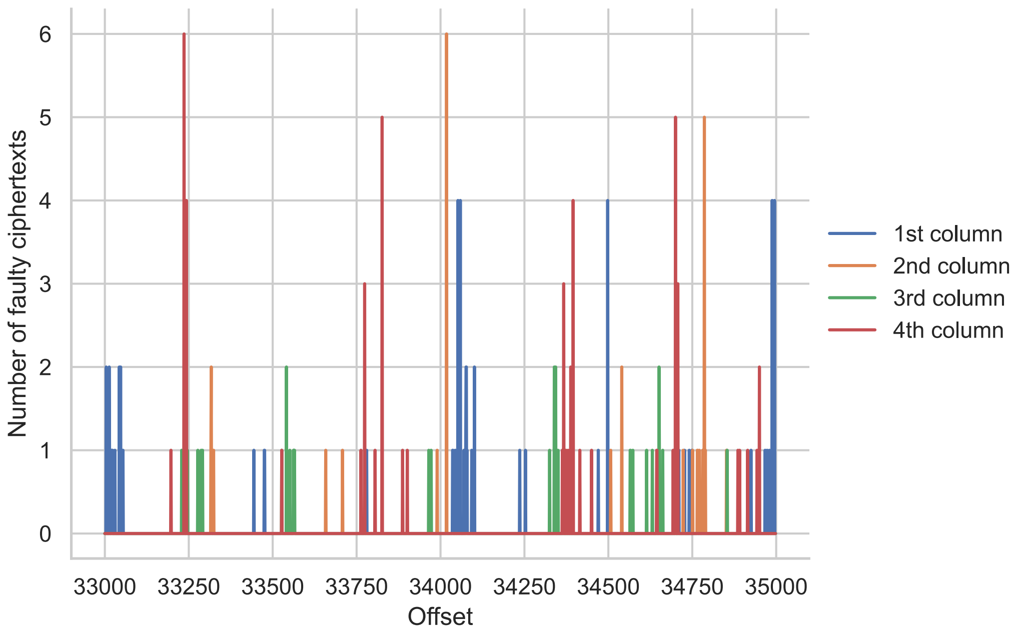
5.5.2 Results

By performing the steps in the implementation section, we successfully recovered the AES encryption key. We also collected data on glitching success rate. The data are stored in file `cw_nano_graph.npy`. The captured ciphertexts are located in `cw_nano_cts.npy`.

The number of ciphertexts captured by offset is shown in figure 5.8. We inserted ten faults at each offset. The faults are divided into four groups based on the column into which they were inserted.

We also tried to decrease the time required for the attack by using only the offsets obtained from figure 5.8, but the results were inconsistent and the only way to obtain enough ciphertexts was to try all the offsets in the range mentioned earlier, i.e. 33000-35000. This behavior might be due to the high jitter of the glitch insertion mechanism of the ChipWhisperer-Nano.

Figure 5.8 Faults by offset on the ChipWhisperer-Nano



Chapter 6
Conclusion

The goal of this work was to study non-invasive fault injection attacks on microcontrollers and existing voltage glitching tools, to implement a fault injection platform on Cmod S7 capable of voltage glitching and possibly clock glitching, and then to perform an attack using the implemented platform.

In this work, we studied and provided an overview of non-invasive fault injection techniques and attacks. We also provided an overview of the existing voltage glitching tools.

In the practical part, fault injection hardware was implemented in Verilog. The hardware is capable of voltage glitching, clock glitching, inserting multiple glitches, and the ability to insert different types of glitches independently. The voltage glitches are inserted using a crowbar circuit. Next, a firmware for the platform and a computer program were created to enable users to control the platform from their computers.

The implemented platform was successfully tested by means of attacking the AES cipher running on an ATmega8L microcontroller. Both voltage and clock faults successfully induced erroneous results in the encryption process of the AES. With the obtained ciphertexts, we recovered the secret AES key stored on an ATmega8L microcontroller. The attack on AES by injecting faults between the 7th and the 8th *MixColumns* was successful as well as the attack on AES with faults injected between the 8th and the 9th *MixColumns*.

The implemented fault injection platform could be further extended. A trigger module with UART communication eavesdropping could be implemented. Another option could be to modify the implemented platform and software to support a digital-to-analog converter that could be used to control the shapes of the inserted glitches.

Appendix A
Attack on AES by Piret et al.

Here we give the algorithm for the attack on AES proposed by Piret et al. The algorithm A.1 describes how to recover the key, where *cts* is the array of correct ciphertexts, *faulty_cts* is the array with corresponding erroneous ciphertexts, and *Inv* suffix denotes the inverse transformation. Algorithm A.2 shows how to decrease the search space.

Algorithm A.1 Recovery of K_{10} with fault between the 7th and the 8th *MixColumns*. Source: article by Piret et al. [32], modified into pseudocode

```
1:  procedure RECOVERY78PIRET($cts$, $faulty\_cts$)
2:      $D \leftarrow \{\}$
3:      for every 16-byte array $a$ with one non-zero byte do
                ▷ Differences between ciphertexts at the output of $\theta_{R-1}$ layer
4:          $D \leftarrow D \cup$ MixColumns(ShiftRows($a$))
5:      end for
6:      $key\_candidates \leftarrow$ REDUCESEARCHSPACE($cts[0]$, $faulty\_cts[0]$, $cts[1]$, $faulty\_cts[1]$)
7:      for $c$ in $\{0, 4, 8, 12\}$ do                ▷ For every quartet of $K_{10}$ key's bytes
8:          $idx \leftarrow 0$
9:          while $|key\_candidates| > 1$ do           ▷ Until there is only one candidate left
10:             $ct \leftarrow$ ShiftRowsInv($cts[idx]$)   ▷ Reorder bytes by applying inverse of ShiftRows
11:             $faulty\_ct \leftarrow ct$
12:             $faulty\_ct[c \dots c+3] \leftarrow$ ShiftRowsInv($faulty\_cts[idx]$)$[c \dots c+3]$
                ▷ Keep faults only in one column
13:             $key\_candidates' \leftarrow \{\}$
14:             for $key\_guess$ in $key\_candidates$ do
15:                 $d \leftarrow$ SubBytesInv($ct \oplus key\_guess$) $\oplus$ SubBytesInv($faulty\_ct \oplus key\_guess$)
                ▷ $\gamma^{-1} \circ \sigma[key\_guess](ct) \oplus \gamma^{-1}\sigma[key\_guess](faulty\_ct)$
16:                 if $d \in D$ then
17:                     $key\_candidates' \leftarrow key\_candidates' \cup key\_guess$
18:                 end if
19:             end for
20:             $key\_candidates \leftarrow key\_candidates'$
21:             $idx \leftarrow idx + 1$
22:         end while
23:     end for
24:     return ShiftRows($key\_candidates[0]$)         ▷ Correctly arrange the bytes of $K_{10}$
25: end procedure
```

Algorithm A.2 Search space reduction. Source: article by Piret et al. [32], modified into pseudocode

```
procedure ReduceSearchSpace($ct_1$, $faulty\_ct_1$, $ct_2$, $faulty\_ct_2$)
    $D \leftarrow \{\}$
    for every 16-byte array $a$ with one non-zero byte do
                                ▷ Differences between ciphertexts at the output of $\theta_{R-1}$ layer
        $D \leftarrow D \cup$ MixColumns(ShiftRows($a$))
    end for
    $key\_candidates \leftarrow \{\}$                                  ▷ Set of key candidates
    for $c$ in $\{0, 4, 8, 12\}$ do     ▷ For every column to obtain four times four bytes of $K_{10}$
        for $i$ in $\{1, 2\}$ do                          ▷ Transform both ciphertext pairs
            $ct'_i \leftarrow$ ShiftRowsInv($ct_i$)      ▷ Reorder bytes by applying inverse of *ShiftRows*
            $faulty\_ct'_i \leftarrow ct'_i$
            $faulty\_ct'_i[c \dots c+3] \leftarrow$ ShiftRowsInv($faulty\_ct_i$)$[c \dots c+3]$
                                                          ▷ Keep faults only in one column
        end for
        $D' \leftarrow \{d[c : c+1] : d \in D\}$      ▷ Set of values from $D$ restricted to only two bytes
        $L \leftarrow \{\}$                                        ▷ Possible bytes of $K_{10}$
        for $k_c$ in $\{00, \dots FF\}$ do
            for $k_{c+1}$ in $\{00, \dots FF\}$ do ▷ For every first two bytes of a key $K_{10}$ in column $c/4$
                $key\_guess \leftarrow [00, \dots, 00, k_c, k_{c+1}, 00, \dots, 00]$   ▷ Pad with $c$ and $14 - c$ zeroes
                $d_1 \leftarrow$ SubBytesInv($ct'_1 \oplus key\_guess$)) $\oplus$ SubBytesInv($faulty\_ct'_1 \oplus key\_guess$))
                $d_2 \leftarrow$ SubBytesInv($ct'_2 \oplus key\_guess$)) $\oplus$ SubBytesInv($faulty\_ct'_2 \oplus key\_guess$))
                                    ▷ $\gamma^{-1} \circ \sigma[key\_guess](ct'_i) \oplus \gamma^{-1}\sigma[key\_guess](faulty\_ct'_i)$
                if $d_1[c : c+1] \in D'$ and $d_2[c : c+1] \in D'$ then
                                                  ▷ Compare differences with those in $D'$.
                    $L \leftarrow L \cup key\_guess$
                end if
            end for
        end for
        for $j$ in $\{c+2, c+3\}$ do                      ▷ Extend the key guesses to four bytes
            $D' \leftarrow \{d[c : j] : d \in D\}$                ▷ Restrict values in $D$ to $j - c + 1$ bytes
            $L' \leftarrow \{\}$                          ▷ Contains keys from $L$ extended by one byte
            for $guess \in L$ do
                for $k_j$ in $\{00, \dots, FF\}$ do
                    $guess' \leftarrow [00, \dots, 00, guess, k_j, 00, \dots, 00]$   ▷ Pad with $c$ and $15 - j$ zeroes
                    $d_1 \leftarrow$ SubBytesInv($ct'_1 \oplus guess'$)) $\oplus$ SubBytesInv($faulty\_ct'_1 \oplus guess'$))
                    $d_2 \leftarrow$ SubBytesInv($ct'_2 \oplus guess'$)) $\oplus$ SubBytesInv($faulty\_ct'_2 \oplus guess'$))
                    if $d_1[c : j] \in D'$ and $d_2[c : j] \in D'$ then
                        $L' \leftarrow L' \cup guess'$
                    end if
                end for
            end for
            $L \leftarrow L'$
        end for
        $key\_candidates \leftarrow$ concatenate values from $L$ to candidates in $key\_candidates$
    end for
    return $key\_candidates$
    ▷ Bytes of $key\_candidates$ are not in the correct order, but they are not reordered since
    they would be transformed back in the main function
end procedure
```

Appendix B

Tutorial on fault injection using our platform

We also created a tutorial in Python in the form of a Jupyter Notebook. The tutorial offers step-by-step instructions for carrying out an attack on AES using the platform that was implemented in this work. The notebook contains only the fault injection part of the attack because its main purpose is to familiarize users with the tool.

The code is broken up into functions that the user has to fill in to perform the attack. By following the tutorial, the user can learn how to use both voltage glitching and clock glitching. The notebook also lets the user choose between two glitch trigger types. The glitches can be triggered on a reset of a target device or after receiving a trigger signal from the target. The latter allows the user to change the encryption key and the plaintext, which is encrypted by the target.

The first notebook is in a file `FaultInjectionTemplate.ipynb`. There is also a second notebook `FaultInjectionFull.ipynb` with all the missing code filled in.

We were also able to get feedback on the tutorial from a student. We edited task assignments based on the observation that they did not contain enough information.

Appendix C

Build manual

The build process consists of three parts. First, the hardware platform has to be built in Vivado. During the second part, the application for the MicroBlaze is compiled. Then, the compiled application can be loaded onto the FPGA. Optionally, the FPGA's flash is programmed.

The build can be done manually by following the steps in sections C.1 and C.2, or by running the provided `build_all.py` script. The two programming sections are not automated.

The build process was tested with version 2023.2 of Vivado and the same version of Vitis.

C.1 Building hardware

To generate the platform, follow these steps:

1. Open Vivado (preferably version 2023.2)
2. Click on *Tools* in the top navigation bar.
3. Select *Run Tcl Script...*
4. Select *project_glitch.tcl* and proceed with *OK*.
5. After the project is created, select in *Flow Navigator*, which is on the left side of the screen, *Generate Bitstream* under *PROGRAM AND DEBUG*.
6. *Synthesis is Out-of-date* dialog window might pop up. Press *Yes*.
7. Then a *Launch Runs* window appears.
8. Optionally, change in the window *Number of jobs* to speed up the synthesis process. Then click *OK*.
9. When the synthesis and implementation finish, another dialog window with header *Device Image Generation Completed* will appear, and press *Cancel*.

Now, the platform is ready to be exported for use in Vitis:

1. In the top navigation bar of Vivado, select *File*, *Export*, *Export Hardware...*.
2. Click *Next* in the *Export Hardware Platform* window.
3. As output select option *Include bitstream*.

4. Continue by clicking *Next*.
5. In the *Files* window, the file can be named and it's location can be changed.
6. Then click on *Finish* button.

   The hardware platform is exported, and Vivado can be closed.

C.2 Building application

First, the hardware platform has to be added:

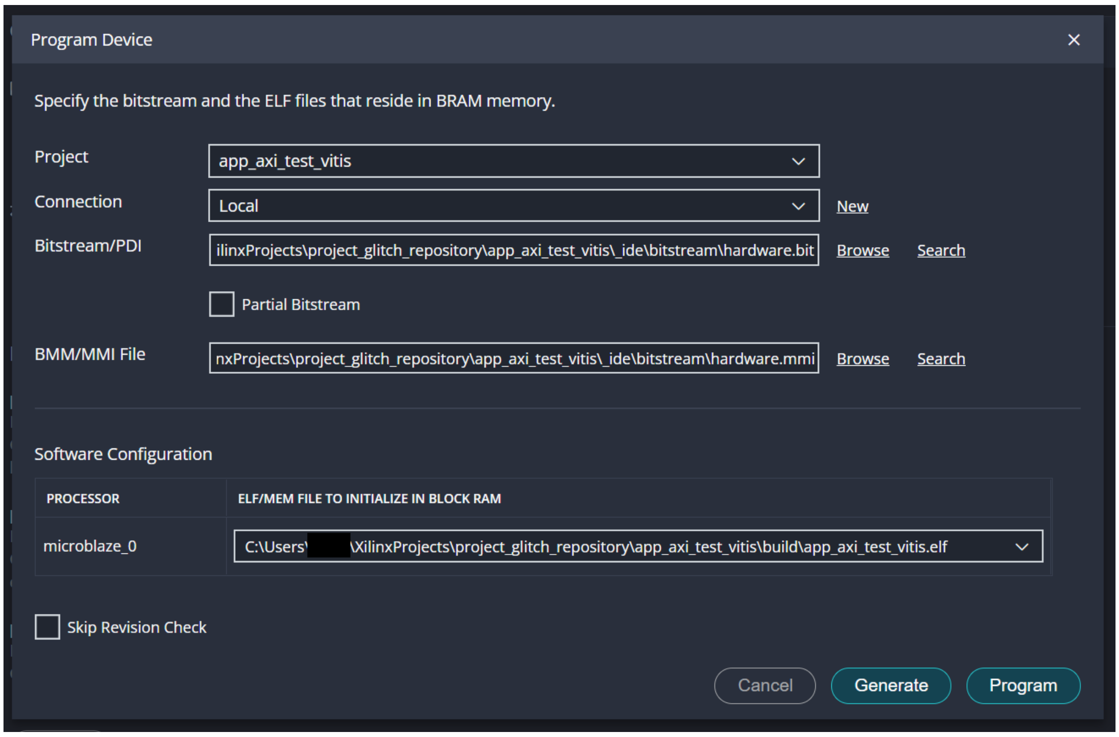
1. Open Vitis.
2. In the top navigation, select *Open Workspace*.
3. Select the parent directory of app_glitch.
4. In the top navigation bar, select *File*, *New Component*, *Platform*.
5. Name the platform and click on *Next*.
6. In the *Hardware Desing (XSA)* field, choose the .xsa file created in the previous section.
7. Click *Next* twice and then click on *Finish*.
8. Locate *FLOW* in the left third of the screen.
9. Click on *Build* in the *FLOW* window.

   Then, the application has to be created:

1. In the top navigation bar, select *File*, *New Component*, *Application*.
2. Name the application and click on *Next*.
3. Select the platform from the previous step and proceed by clicking *Next*.
4. Click *Next* again and then *Finish*.
5. Copy files from *app_glitch\src* into *name of the application\src*.
6. Locate *FLOW* in the left third of the screen.
7. Click on *Build* in the *FLOW* window.

C.3 Device programming

1. Open Vitis.
2. Open the workspace with app_glitch_vitis.
3. Connect the Digilent Cmod S7 to the PC.
4. The built application can be run by pressing *Run* in the *FLOW* menu under the *Build* button.



Program Device

Specify the bitstream and the ELF files that reside in BRAM memory.

Project	app_axi_test_vitis
Connection	Local
Bitstream/PDI	ilinxProjects\project_glitch_repository\app_axi_test_vitis_ide\bitstream\hardware.bit
	☐ Partial Bitstream
BMM/MMI File	nxProjects\project_glitch_repository\app_axi_test_vitis_ide\bitstream\hardware.mmi

Software Configuration

PROCESSOR	ELF/MEM FILE TO INITIALIZE IN BLOCK RAM
microblaze_0	C:\Users\[redacted]\XilinxProjects\project_glitch_repository\app_axi_test_vitis\build\app_axi_test_vitis.elf

☐ Skip Revision Check

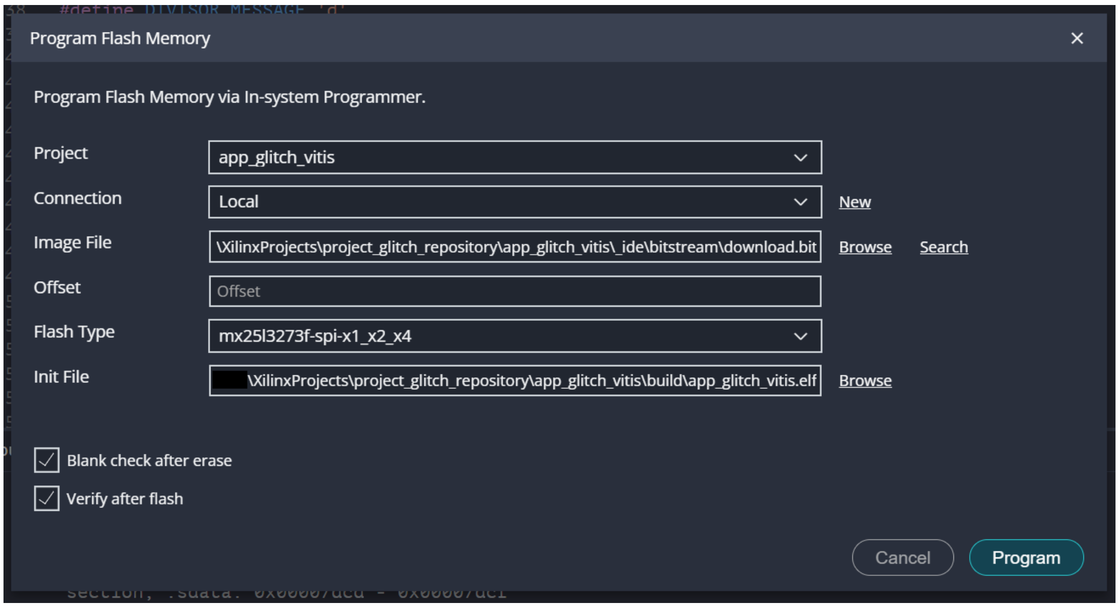
Figure C.1 Program Device window

.

C.4 Flash programming

The flash can be programmed to permanently store the hardware configuration and the program in a non-volatile memory of the FPGA. Follow these steps to store the program in the FPGA's flash memory:

1. Open Vitis and the same workspace as in the previous sections.
2. Connect the Digilent Cmod S7 to the PC.
3. In the top navigation bar, select *Vitis*, *Program Device*.
4. Choose app_glitch_vitis in the *Project* dropdown menu.
5. Select *hardware.bit* in the *Bitstream/PDI* field.
6. Select *hardware.mmi* as the *BMM/MMI File*.
7. Select *app_glitch_vitis.elf* in the *microblaze_0* field.
8. The window should look like the one in the figure C.1. The content of the *Project* field should contain the name of the application from section C.2, and might differ from the one in the picture.
9. Click on *Generate*.
10. Wait for Vitis to generate *download.bit* file, and close the *Program Device* window.
11. Then, select *Vitis*, *Program Flash* in the top navigation bar.


Program Flash Memory

Program Flash Memory via In-system Programmer.

Project	app_glitch_vitis
Connection	Local
Image File	\XilinxProjects\project_glitch_repository\app_glitch_vitis_ide\bitstream\download.bit
Offset	Offset
Flash Type	mx25l3273f-spi-x1_x2_x4
Init File	[redacted]\XilinxProjects\project_glitch_repository\app_glitch_vitis\build\app_glitch_vitis.elf

☑ Blank check after erase

☑ Verify after flash

Figure C.2 Program Flash window

12. Select app_glitch_vitis in the *Project* field.
13. Select *download.bit* file as *Image File*.
14. Choose `mx25l3273f-spi-x1_x2_x4` as *Flash Type*.
15. Select app_glitch_vitis.elf file as the *Init File*.
16. Check *Blank check after erase* and *Verify after flash* checkboxes.
17. The window should look like the one in the figure C.2. Again, the *Project* field's content might be different.
18. Click on *Program* to program the flash.
19. Disconnect the Digilent Cmod S7 from to PC after the flash programming is done.

Bibliography

1. STANDAERT, François-Xavier. Introduction to Side-Channel Attacks. In: *Secure Integrated Circuits and Systems*. Ed. by VERBAUWHEDE, Ingrid M.R. Boston, MA: Springer US, 2010, pp. 27–42. ISBN 978-0-387-71829-3. Available from DOI: `10.1007/978-0-387-71829-3_2`.
2. KOCHER, Paul C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Berlin, Heidelberg: Springer Berlin Heidelberg, [n.d.], pp. 104–113. Advances in Cryptology — CRYPTO '96. ISSN 0302-9743.
3. KOCHER, Paul; JAFFE, Joshua; JUN, Benjamin. Differential Power Analysis. In: Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397. Advances in Cryptology — CRYPTO' 99. ISSN 0302-9743.
4. BIHAM, Eli; SHAMIR, Adi. Differential fault analysis of secret key cryptosystems. In: KALISKI, Burton S. (ed.). *Advances in Cryptology — CRYPTO '97*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 513–525. ISBN 978-3-540-69528-8.
5. LI, Yang; OHTA, Kazuo; SAKIYAMA, Kazuo. New Fault-Based Side-Channel Attack Using Fault Sensitivity. *IEEE transactions on information forensics and security*. 2012, vol. 7, no. 1, pp. 88–97. ISSN 1556-6013.
6. AGRAWAL, Dakshi; ARCHAMBEAULT, Bruce; RAO, Josyula R.; ROHATGI, Pankaj. The EM Side—Channel(s). In: Berlin, Heidelberg: Springer Berlin Heidelberg, [n.d.], pp. 29–45. Cryptographic Hardware and Embedded Systems - CHES 2002. ISSN 0302-9743.
7. ASONOV, D.; AGRAWAL, R. Keyboard acoustic emanations. In: *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*. 2004, pp. 3–11. Available from DOI: `10.1109/SECPRI.2004.1301311`.
8. AMIEL, Frederic; VILLEGAS, Karine; FEIX, Benoit; MARCEL, Louis. Passive and Active Combined Attacks: Combining Fault Attacks and Side Channel Analysis. In: *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*. 2007, pp. 92–102. Available from DOI: `10.1109/FDTC.2007.12`.
9. SKOROBOGATOV, Sergei P.; ANDERSON, Ross J. Optical Fault Induction Attacks. In: KALISKI, Burton S.; KOÇ, çetin K.; PAAR, Christof (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2002*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–12. ISBN 978-3-540-36400-9.
10. CLAVIER, Christophe. Secret External Encodings Do Not Prevent Transient Fault Analysis. In: PAILLIER, Pascal; VERBAUWHEDE, Ingrid (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 181–194. ISBN 978-3-540-74735-2.

11. YEN, Sung-Ming; JOYE, M. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*. 2000, vol. 49, no. 9, pp. 967–970. Available from DOI: `10.1109/12.869328`.

12. BARENGHI, Alessandro; BREVEGLIERI, Luca; KOREN, Israel; NACCACHE, David. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. *Proceedings of the IEEE*. 2012, vol. 100, no. 11, pp. 3056–3076. Available from DOI: `10.1109/JPROC.2012.2188769`.

13. ZUSSA, Loïc; DUTERTRE, Jean-Max; CLÉDIÈRE, Jessy; TRIA, Assia. Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism. In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. 2013, pp. 110–115. Available from DOI: `10.1109/IOLTS.2013.6604060`.

14. KORAK, Thomas; HOEFLER, Michael. On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms. In: IEEE, 2014, pp. 8–17. ISBN 978-1-4799-6292-1.

15. RAZAVI, Behzad. *Fundamentals of Microelectronics*. ”2”. Wiley, 2014. ISBN 978-1-118-15632-2.

16. HUTTER, Michael; SCHMIDT, Jorn-Marc; PLOS, Thomas. Contact-based fault injections and power analysis on RFID tags. In: *2009 European Conference on Circuit Theory and Design*. 2009, pp. 409–412. Available from DOI: `10.1109/ECCTD.2009.5275012`.

17. O’FLYNN, Colin; CHEN, Zhizhang (David). ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research. In: PROUFF, Emmanuel (ed.). *Constructive Side-Channel Analysis and Secure Design*. Cham: Springer International Publishing, 2014, pp. 243–260. ISBN 978-3-319-10175-0.

18. BOZZATO, Claudio; FOCARDI, Riccardo; PALMARINI, Francesco, et al. Shaping the glitch: optimizing voltage fault injection attacks. *IACR transactions on cryptographic hardware and embedded systems*. 2019, vol. 2019, no. 2, pp. 199–224.

19. ZUSSA, Loic; DUTERTRE, Jean-Max; CLEDIERE, Jessy; ROBISSON, Bruno. Analysis of the fault injection mechanism related to negative and positive power supply glitches using an on-chip voltmeter. In: *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2014, pp. 130–135. Available from DOI: `10.1109/HST.2014.6855583`.

20. DJELLID-OUAR, A.; CATHEBRAS, G.; BANCEL, F. Supply voltage glitches effects on CMOS circuits. In: *International Conference on Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006.* 2006, pp. 257–261. Available from DOI: `10.1109/DTIS.2006.1708651`.

21. SCHMIDT, Jörn-Marc; HUTTER, Michael. 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*. Verlag der Technischen Universität Graz, 2007, pp. 61–67. ISBN 978-3-902465-87-0.

22. DUMONT, Mathieu; LISART, Mathieu; MAURINE, Philippe. Electromagnetic Fault Injection : How Faults Occur. In: *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2019, pp. 9–16. Available from DOI: `10.1109/FDTC.2019.00010`.

23. GOVINDAVAJHALA, S.; APPEL, A.W. Using memory errors to attack a virtual machine. In: *2003 Symposium on Security and Privacy, 2003.* 2003, pp. 154–165. Available from DOI: `10.1109/SECPRI.2003.1199334`.

24. HUTTER, Michael; SCHMIDT, Jörn-Marc. The Temperature Side Channel and Heating Fault Attacks. In: FRANCILLON, Aurélien; ROHATGI, Pankaj (eds.). *Smart Card Research and Advanced Applications*. Cham: Springer International Publishing, 2014, pp. 219–235. ISBN 978-3-319-08302-5.

25. KIM, Yoongu; DALY, Ross; KIM, Jeremie; FALLIN, Chris; LEE, Ji Hye; LEE, Donghyuk; WILKERSON, Chris; LAI, Konrad; MUTLU, Onur. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. *SIGARCH Comput. Archit. News*. 2014, vol. 42, no. 3, pp. 361–372. ISSN 0163-5964. Available from DOI: `10.1145/2678373.2665726`.

26. SUBIDH ALI, SK; MUKHOPADHYAY, Debdeep; TUNSTALL, Michael. Differential fault analysis of AES: Towards reaching its limits. *Journal of Cryptographic Engineering*. 2012, vol. 3. Available from DOI: `10.1007/s13389-012-0046-y`.

27. BIHAM, Eli; SHAMIR, Adi. Differential fault analysis of secret key cryptosystems. In: KALISKI, Burton S. (ed.). *Advances in Cryptology — CRYPTO '97*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 513–525. ISBN 978-3-540-69528-8.

28. NASHIMOTO, Shoei; HOMMA, Naofumi; HAYASHI, Yu-ichi; TAKAHASHI, Junko; FUJI, Hitoshi; AOKI, Takafumi. Buffer overflow attack with multiple fault injection and a proven countermeasure. *Journal of cryptographic engineering*. 2017, vol. 7, no. 1, pp. 35–46. ISSN 2190-8508.

29. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Advanced Encryption Standard (AES)* [online]. 2023. [visited on 2024-04-01]. Available from DOI: `https://doi.org/10.6028/NIST.FIPS.197-upd1`.

30. DUSART, Pierre; LETOURNEUX, Gilles; VIVOLO, Olivier. Differential Fault Analysis on A.E.S. In: ZHOU, Jianying; YUNG, Moti; HAN, Yongfei (eds.). *Applied Cryptography and Network Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 293–306. ISBN 978-3-540-45203-4.

31. DUSART, P.; LETOURNEUX, G.; VIVOLO, O. *Differential Fault Analysis on A.E.S.* [Online]. 2003. [visited on 2024-04-24]. Available from: `https://eprint.iacr.org/2003/010`.

32. PIRET, Gilles; QUISQUATER, Jean-Jacques. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad. In: WALTER, Colin D.; KOÇ, Çetin K.; PAAR, Christof (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2003*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 77–88. ISBN 978-3-540-45238-6.

33. SHCHAVLEVA, Marina. *Active attack: Differential Fault Analysis Example* [online]. 2021. [visited on 2024-04-30]. Available from: `https://courses.fit.cvut.cz/NI-HWB/tutorials/06/dfa.html`.

34. WIERSMA, Nils; PAREJA, Ramiro. Safety != Security: On the Resilience of ASIL-D Certified Microcontrollers against Fault Injection Attacks. In: *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2017, pp. 9–16. Available from DOI: `10.1109/FDTC.2017.15`.

35. XILINX, INC. *7 Series FPGAs Data Sheet: Overview* [online]. 2020. [visited on 2024-05-08]. Available from: `https://docs.amd.com/v/u/en-US/ds180_7Series_Overview`.

36. ARM LIMITED. *AMBA® AXI Protocol Specification* [online]. 2023. [visited on 2024-04-20]. Available from: `https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/IHI0022H_amba_axi_protocol_spec.pdf`.

37. FRENZEL, Louis E. *Handbook of Serial Communications Interfaces: A Comprehensive Compendium of Serial Digital Input/Output (I/o) Standards*. 1st ed. Oxford: Elsevier Science & Technology, 2015. ISBN 978-0-128-00629-0.

38. RANKL, Wolfgang; EFFING, Wolfgang. *Smart Card Handbook*. 4th ed. Chichester, West Sussex, U.K: Wiley, 2010. ISBN 978-0-470-74367-6.

39. NEWAE TECHNOLOGY INC. *Auto — NewAE Technology — newae.com* [online]. © 2020. [visited on 2024-03-25]. Available from: `https://www.newae.com/auto`.
40. NEWAE TECHNOLOGY INC. *CHIPWHISPERER — NewAE Technology — newae.com* [online]. © 2020. [visited on 2024-03-25]. Available from: `https://www.newae.com/chipwhisperer`.
41. NEWAE TECHNOLOGY INC. *CW1101: ChipWhisperer-Nano* [online]. 2019. [visited on 2024-03-25]. Available from: `https://media.newae.com/datasheets/NAE-CW1101_datasheet.pdf`.
42. NEWAE TECHNOLOGY INC. *CW1173: ChipWhisperer-Lite* [online]. 2023. [visited on 2024-03-25]. Available from: `https://media.newae.com/datasheets/NAE-CW1173_datasheet.pdf`.
43. NEWAE TECHNOLOGY INC. *CW1200 ChipWhisperer-Pro* [online]. © 2015–2021. [visited on 2024-04-22]. Available from: `https://rtfm.newae.com/Capture/ChipWhisperer-Pro/`.
44. NEWAE TECHNOLOGY INC. *Introduction to ChipWhisperer Husky* [online]. 2023. [visited on 2024-03-25]. Available from: `https://github.com/newaetech/chipwhisperer-jupyter/blob/master/demos/husky/01%20-%20Introduction%20to%20ChipWhisperer-Husky.ipynb`.
45. RISCURE. *Spider* [online]. 2017. [visited on 2024-04-30]. Available from: `https://getquote.riscure.com/picdb/filedb/3792/spider_datasheet_1.pdf`.
46. RISCURE. *VC Glitcher* [online]. 2011. [visited on 2024-04-22]. Available from: `https://getquote.riscure.com/picdb/filedb/3792/VC%20glitcher%20datasheet.pdf`.
47. KASPER, Timo; OSWALD, David; PAAR, Christof. A Versatile Framework for Implementation Attacks on Cryptographic RFIDs and Embedded Devices. *Transactions on Computational Science*. 2010, vol. 10, pp. 100–130. ISBN 978-3-642-17498-8. Available from DOI: `10.1007/978-3-642-17499-5_5`.
48. OSWALD, David. *Implementation Attacks: From Theory to Practice*. 2013. Available from DOI: `10.13140/2.1.4928.8484`. PhD thesis.
49. SASS, Marvin; MITEV, Richard; AHMAD-REZA, Sadeghi. *Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M*. 2023. Available also from: `http://ezproxy.techlib.cz/login?url=https://www.proquest.com/working-papers/oops-i-glitched-again-how-multi-glitch-glitching/docview/2776857861/se-2`.
50. NEWAE TECHNOLOGY INC. *Overview & Comparison* [online]. © 2015–2021. [visited on 2024-03-25]. Available from: `https://rtfm.newae.com/Capture/`.
51. RISCURE. *Spider* [online]. [N.d.]. [visited on 2024-04-30]. Available from: `https://www.riscure.com/products/spider/`.
52. NEWAE TECHNOLOGY INC. *Scope API* [online]. © 2024. [visited on 2024-04-29]. Available from: `https://chipwhisperer.readthedocs.io/en/latest/scope-api.html`.
53. NEWAE TECHNOLOGY INC. *Overview — ChipWhisperer 5.7.0 documentation* [online]. © 2024. [visited on 2024-05-14]. Available from: `https://chipwhisperer.readthedocs.io/en/latest/getting-started.html`.
54. DIGILENT. *Cmod S7 Reference Manual* [online]. 2023. [visited on 2024-05-14]. Available from: `https://digilent.com/reference/programmable-logic/cmod-s7/reference-manual`.
55. ADVANCED MICRO DEVICES, INC. *Vivado Overview* [comp. software]. © 2024. [visited on 2024-05-01]. Available from: `https://www.xilinx.com/products/design-tools/vivado.html`.

56. ADVANCED MICRO DEVICES, INC. *MicroBlaze* [online]. © 2024. [visited on 2024-05-08]. Available from: `https://www.xilinx.com/products/intellectual-property/microblazecore.html`.
57. ADVANCED MICRO DEVICES, INC. *Vitis Unified Software Platform* [comp. software]. © 2024. [visited on 2024-05-01]. Available from: `https://www.xilinx.com/products/design-tools/vitis.html`.
58. LIECHTI, Chris. *Pyserial/pyserial: Python serial port access library* [comp. software]. 2023. [visited on 2024-04-22]. Available from: `https://github.com/pyserial/pyserial`.
59. ATMEL CORPORATION. *ATmega8/L datasheet* [online]. 2013. [visited on 2024-04-09]. Available from: `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2486-8-bit-AVR-microcontroller-ATmega8_L_datasheet.pdf`.

Contents of the attachment

- `README.txt` .... brief description of the contents
- `LICENSE.txt` .... license
- `src`
  - `attacks_on_aes` .... implementation of attacks on the AES
  - `cmod` .... source code of our fault injection platform implemented on Cmod S7
    - `cmod_sw` .... software source code of our platform
    - `cmod_hw_and_fw` .... firmware and software source code of our platform
    - `notebooks` .... tutorial and examples of how to use our platform
  - `thesis` .... source code of the text in LaTeX
- `collected_data` .... collected ciphertexts and data on glitching success rate
- `firmware_mega8_aes` .... ATmega8L firmware and AES code
- `cw_nano` .... firmware for ChipWhisperer-Nano and its target
- `text`
  - `thesis.pdf` .... text of the thesis in PDF format