Development of an Integrated Environment for Side Channel Analysis and Fault Injection

David Oswald

Matr. Nr.: 108004206932

September 11, 2009

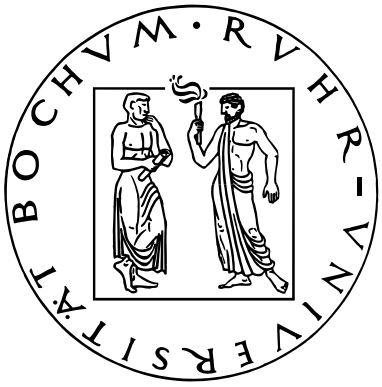
Diplomarbeit

Lehrstuhl für Embedded Security
Prof. Dr.-Ing. Christof Paar

Advised by:

Dipl.-Ing. Timo Kasper

Ruhr-Universität Bochum



Statement

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

Hiermit versichere ich, dass ich meine Diplomarbeit selbst verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und Zitate kenntlich gemacht habe.

David Oswald

Bochum, September 11, 2009

Contents

1. Introduction **1**
- 1.1. Implementation Attacks . . . 1
- 1.2. Embedded Systems . . . 2
  - 1.2.1. Microcontroller . . . 2
  - 1.2.2. Smartcard . . . 3
  - 1.2.3. Radio Frequency Identification . . . 3
- 1.3. Outline of this Thesis . . . 4

2. A Survey of Side-Channel Analysis and Fault Injection Attacks **5**
- 2.1. Side-Channel Analysis . . . 5
  - 2.1.1. Simple Power Analysis . . . 5
  - 2.1.2. Differential Power Analysis . . . 6
- 2.2. Fault Injection . . . 7
  - 2.2.1. General Parameters of Fault Injection . . . 7
  - 2.2.2. Types of Physical Faults in Integrated Circuits . . . 8
  - 2.2.3. Methods for Fault Injection Attacks on Cryptographic Algorithms 10
- 2.3. Combined Attacks . . . 20

3. A Framework for Side-Channel Measurements and Fault Injection Attacks **21**
- 3.1. Development Toolchain . . . 22
- 3.2. Configuration Files . . . 22
- 3.3. Communication Modules . . . 23
  - 3.3.1. ISO 7816 Smartcard Reader . . . 23
  - 3.3.2. Contactless Smartcard Reader . . . 24
  - 3.3.3. Arbitrary Parallel/Serial Communication . . . 25
- 3.4. Measurement Modules . . . 26
  - 3.4.1. Oscilloscope . . . 26
  - 3.4.2. Analogue Pre-Processing . . . 31
  - 3.4.3. Framework Class for Measurement Applications . . . 32
  - 3.4.4. Framework Class for Evaluation and Processing Applications . . . 34
- 3.5. FPGA-based Module for Fault Injection . . . 37
  - 3.5.1. Overall System Structure . . . 38
  - 3.5.2. Power Fault Module . . . 45
  - 3.5.3. Clock Fault Module . . . 48

4. Practical Attacks **55**
4.1. Side-Channel Analysis 55
4.1.1. Microcontroller-based Smartcard 55
4.1.2. Commercial Contactless Smartcard 57
4.2. Fault Injection Attacks 65
4.2.1. Single Faults 65
4.2.2. Multiple Faults 69

5. Conclusion **73**
5.1. Summary 73
5.2. Future Work 74

A. Algorithms **75**

B. Framework Documentation **83**
B.1. Class List 83
B.2. Class Hierachy 86

C. Schematics **89**

D. Bibliography **93**

List of Figures

1.1. Classification of implementation attacks. 2

2.1. First subkey addition and S-Box of AES for first plaintext byte (based on figure in [Wik09]). 18

3.1. Overall system structure for combined side-channel analysis and fault injection. 21
3.2. Block diagram for reduction of RFID reader field influence. 32
3.3. Program flow of measurement framework application. 33
3.4. Program flow of measurement framework application. 34
3.5. Graphical representation of an example processing chain. 37
3.6. FPGA structure overview. 38
3.7. Structure of power fault module. 45
3.8. Output stage of power fault module, revision 1. 46
3.9. Full-scale 10 ns negative voltage glitch at DAC output, x10 probe. 47
3.10. Full-scale 10 ns negative voltage glitch at first amplifier output, x10 probe. 47
3.11. Output stage of power fault module, revision 2. 47
3.12. Full-scale 10 ns voltage glitch at amplifier output, module revision 2, x10 probe. 48
3.13. Full-scale 10 ns voltage glitch after transistor output stage, module revision 2, x10 probe. 48
3.14. Clock fault signal $o_1 = \overline{clk} \wedge \overline{clk_s}$ for shortening clock cycles. 49
3.15. Clock fault signal $o_3 = clk \vee \underline{clk_s}$ for stretching clock cycles. 49
3.16. Clock fault signal $o_2 = clk \wedge \overline{clk_s}$ for short positive pulses. 50
3.17. Clock fault signal $o_4 = \overline{clk} \vee clk_s$ for short negative pulses. 50
3.18. Clock signal shifting and prescaling on FPGA. 51
3.19. Clock fault on 16.67 MHz clock signal, x10 probe, 20 ns/time division. . . 52
3.20. 16.67 MHz clock signal after output buffer, x10 probe. 52
3.21. 50 MHz clock signal after output buffer, x10 probe. 52

4.1. Correlation after $L = 300$ traces for first byte in first round of AES, Hamming weight. 56
4.2. Correlation after $L = 300$ traces for first byte in first round of AES, Hamming weight (zoomed). 56
4.3. Exerpt of the authentication protocol relevant for an attack. 58

4.4. Raw trace of 3DES encryption with analogue filter (zoomed). 59
4.5. Raw trace of 3DES encryption without analogue filter. 59
4.6. Block diagram of incoherent digital amplitude demodulator. 60
4.7. Demodulated trace (50 kHz - 2 MHz) of 3DES encryption with analogue filter. 60
4.8. Demodulated trace (50 kHz - 2 MHz) of 3DES encryption without analogue filter. 60
4.9. Correlation coefficients for plaintext bytes (before targeted 3DES encryption) after 5,000 traces, Hamming Weight. 61
4.10. Correlation coefficients for ciphertext bytes (after targeted 3DES encryption) after 2,000 traces, Hamming Weight. 61
4.11. Overview over operations in amplitude-demodulated trace. 62
4.12. Part of trace with 3DES encryption, filtered with $f_{lowpass} = 8$ MHz, $f_{highpass} = 50$ kHz. 62
4.13. Correlation coefficients for first DES, first round, after 1,000,000 traces, $f_{lowpass} = 8$ MHz, $f_{highpass} = 50$ kHz. 63
4.14. Correlation coefficients for second DES, first round, after 1,000,000 traces, $f_{lowpass} = 8$ MHz, $f_{highpass} = 50$ kHz. 65
4.15. Parameters characterising a single negative power glitch. 67
4.16. Waveform of reset after fault injection on PIC16F687. 68
4.17. Waveform of unsuccessful fault injection on PIC16F687. 68
4.18. Waveform of successful fault injection on PIC16F687, `PIN_STATUS`. . . . 69
4.19. Waveform of successful fault injection on PIC16F687, `PIN_STATUS_2`. . . 69
4.20. Parameters characterising a double negative power glitch. 71
4.21. Waveform of successful multiple fault injection on PIC16F687, `PIN_STATUS_2`. 71

C.1. Schematic of power fault module, revision 1. 90
C.2. Schematic of power fault module, revision 2. 91
C.3. Schematic of clock fault module. 92

List of Abbreviations

ADC	Analogue-Digital Converter
ADPU	Application Protocol Data Unit Command
AES	Advanced Encryption Standard
API	Application Programming Interface
CPA	Correlation Power Analysis
CRT	Chinese Remainder Theorem
DAC	Digital-Analogue Converter
DCM	Digital Clock Manager
DEMA	Differential Electro-Magnetic Analysis
DES	Data Encryption Standard
DIP	Dual Inline Package
DL	Discrete Logarithm
DPA	Differential Power Analysis
DSO	Digital Storage Oscilloscope
DUT	Device Under Test
EC	Elliptic Curve
EC DSA	Elliptic Curve Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
EEPROM	Electrically Erasable Programmable Read-Only Memory
EM	Electro-Magnetic
FI	Fault Injection
FPGA	Field Programmable Gate Array
gcc	GNU Compiler Collection
HF	High Frequency
I/O	Input/Output
IC	Integrated Circuit
JTAG	Joint Test Action Group
LF	Low Frequency
NC	Not Connected
NVM	Non-volatile Memory
OP	Operational Amplifier
PACA	Passive Active Combined Attack
PCB	Printed Circuit Board
PCD	Proximity Coupling Device

PICC	Proximity Integrated Circuit Card
RF	Radio Frequency
RFID	Radio Frequency Identification
RISC	Reduced Instruction Set Computing
RNG	Random Number Generator
SCA	Side-Channel Analysis
SNR	Signal-to-Noise Ratio
SOIC	Small-Outline Integrated Circuit
SPA	Simple Power Analysis
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver Transmitter
UHF	Ultra High Frequency
VHDL	Very High Speed Integrated Circuit Hardware Description Language

List of Algorithms

1. RSA Signature Creation with CRT Optimisation. 11
2. Right-to-left double-and-add algorithm. 15
3. Final round of DES. 16
4. Final round of AES. 17
5. Pre-whitenining step of AES. 18

6. Bellcore fault attack on RSA with CRT. 75
7. Lenstra fault attack on RSA with CRT. 75
8. Fault attack on RSA without CRT. 76
9. Fault analysis of EC multiplication with faulty input point. 77
10. Fault analysis of EC with faults during the multiplication. 78
11. Differential fault analysis of final round of DES. 79
12. Differential fault analysis of final round of AES. 80
13. Fault analysis of pre-whitening step of AES. 81

1. Introduction

In the past few years, the paradigm of computing has shifted: *Embedded systems* have become omnipresent, enabling a vast variety of different mobile applications, ranging from electronic payment and contactless identification to advanced car control systems. Due to the specific requirements of a given usage scenario, the details of the actual implementation heavily depend on the respective application.

In many cases, however, it is vital that the system fulfils certain *security targets*, i.e., the privacy of the exchanged data, the authenticity of a participant or the integrity of the stored information. Thus, *cryptographic* techniques are applied to meet these requirements. In turn, adversaries trying to bypass these protections rely on some form of *cryptanalysis*, exploiting weaknesses in any part of the realisation. In this thesis, we address the susceptibility of embedded systems towards the important class of *implementation attacks*.

1.1. Implementation Attacks

Before reviewing device types that are common targets, we focus on the class of attacks we aim to examine. Generally, implementation attacks cover all forms of attacks against cryptographical devices relying on the *physical realisation* of the algorithms in hardware or software, rather than exclusively on abstract *mathematical properties* and resulting theoretical weaknesses. A classification according to [Paa06] is given in Fig. 1.1.

A *Passive attack* or *Side-Channel Analysis* (SCA) is conducted by monitoring the device while it performs a cryptographic operation, e.g., by recording data-dependent variations of the execution time (*timing attack*), the power consumption (*power analysis*) or the *Electro-Magnetic* (EM) emanation (*electro-magnetic analysis*), respectively. In contrast, *active attacks* imply the (permanent or temporary) modification of any part of the physical implementation, e.g., to gather information on the internal processes (*reverse engineering*) or to cause incorrect execution by means of *Fault Injection* (FI), enabling further analysis and the recovery of cryptographic secrets.

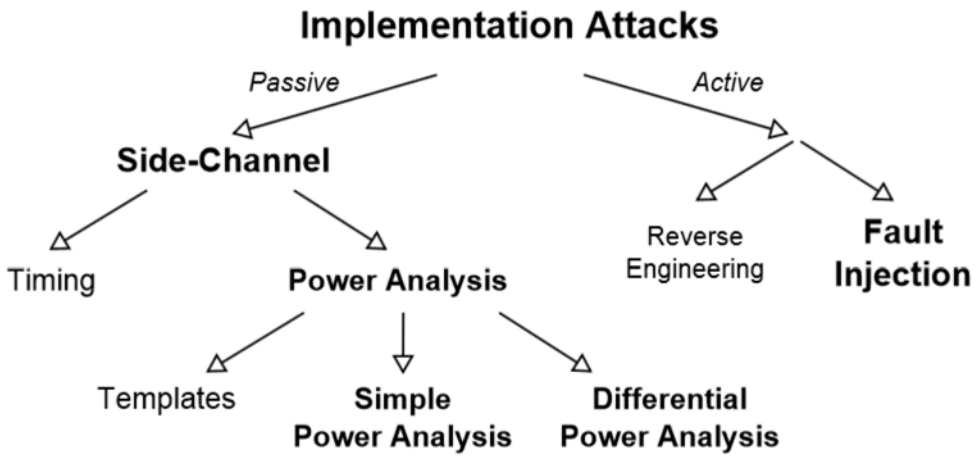


Figure 1.1.: Classification of implementation attacks.

1.2. Embedded Systems

Having specified the considered type of attack, we briefly present some important instances of embedded systems that are of particular interest due to their widespread use and flexibility with respect to possible applications.

1.2.1. Microcontroller

A *microcontroller* is a user-programmable processor extended by special on-chip hardware for typical embedded use-cases, such as serial or parallel communication, measurement, control or the acceleration of cryptographic operations, e.g., data encryption. In contrast to processors employed in desktop computers, microcontrollers are often optimised for low power consumption and minimum cost.

Therefore, the hardware implementation is constrained with respect to the chip area and consequently the number of available gates. Microcontrollers thus often feature a small internal register and bus width, e.g., 8 or 16 bit and run at clock frequencies much lower than those of modern general purpose processors.

Due to their flexibility, microcontrollers have become a popular choice for many tasks in security-sensitive mobile applications, such as car immobilisers, driving control systems or physical access control solutions. Consequently, the implementations usually include cryptographic means to protect against, e.g., counterfeit (cloning) or unauthorised access. Yet, past experience [EKM+08] has shown that the countermeasures with

respect to implementation attacks are often insufficient or missing. Hence, the evaluation whether such attacks are feasible is mandatory to guarantee the long-term security of any newly developed system.

1.2.2. Smartcard

A *smartcard* is a device specifically designed to protect sensitive information and its exchange in high-security scenarios, e.g., payment or the identification of individuals [BSI]. The actual chip is usually contained in a plastic package, as specified in [ISO03]. Smartcards often incorporate a microcontroller to realise control and interface logic and additional hardware accelerators for common cryptographic primitives (e.g., symmetric and asymmetric ciphers or hash functions) with appropriate countermeasures against implementation attacks. Note that, however, the provided level of security again depends on the maximum tolerable cost of the device and thus heavily varies with the targeted application.

1.2.3. Radio Frequency Identification

Radio Frequency Identification (RFID) covers all forms of — most often strictly constrained — devices that are attached to some physical entity and wirelessly respond to a reader to provide information regarding this entity. An RFID device is often called a *tag*.

Popular use-cases include the tracking and identification of assets on different levels, i.e., marking a collection in a container or individual objects, which is usually realised with *Ultra High Frequency* (UHF) tags operating in the range of approx. 400 MHz up to 3 GHz. These systems today rarely involve cryptographic protocols and are not explicitly considered in this thesis.

Rather, we focus on *High Frequency* (HF) and *Low Frequency* (LF) systems running at 13.56 MHz and 100 - 150 kHz, respectively. Especially HF devices are a popular choice for mass-market applications like contactless payment and public transport fare collection. The first generation of these tags bases on proprietary ciphers that turned out to be vulnerable to conventional mathematic cryptanalysis [CNO08]. Newer devices apply established algorithms which are assumed to withstand analytical approaches, yet, implementation attacks can enable an adversary to break this protection and endanger the system security. In the context of HF RFIDs according to the ISO 14443 standard [iso01a, iso01b], the tag is referred to as the *Proximity Integrated Circuit Card* (PICC), while the reader is called *Proximity Coupling Device* (PCD).

Considering that most RFID tags are passively powered, i.e., draw their energy from an EM field generated by a reader and must be available at minimum cost, they are

extremely limited with respect to chip area and processing capabilities. Hence, realising countermeasures against implementation attacks – which often increase the circuit complexity – can be infeasible or at least heavily constrained in many practical scenarios. Thus, when testing a given system, the according attacks have to be actually carried out in order to estimate the provided level of security.

1.3. Outline of this Thesis

This thesis is organised as follows: First we give an overview over the current research status with respect to implementation attacks in Chap. 2, where we both focus on power/EM analysis and fault injection attacks (Reverse engineering and timing attacks are not within the scope of this thesis). For fault injection attacks, a structured survey with respect to practical (i.e., ways to induce erroneous behaviour) and theoretical aspects (i.e., methods to exploit faulty operation results in order to recover cryptographic secrets) is provided. We introduce *combined attacks* which can circumvent countermeasures that protect against individual attacks, however, fail when several methods are applied simultaneously.

In Chap. 3, we present our new, unified framework for realising side-channel analysis and fault injection, supporting combinations of both approaches. Here, special emphasis is placed on the development of a *flexible* and *low-cost* environment to demonstrate that a variety of proposed attacks can be put into practice with limited budget.

Currently, there is a lack of practical results on the feasibility of implementation attacks on real-world devices, as the research is often performed by commercial laboratories and not available to the community. To fill this gap and obtain concrete estimates on the protection level of popular products, we apply the developed framework for the analysis of several real-world devices in Chap. 4.

2. A Survey of Side-Channel Analysis and Fault Injection Attacks

To illustrate the scope of the framework developed in this thesis and to present the fundamentals for the practical attacks given in Chap. 4, we start with an overview over side-channel analysis and define important terms and methods. Then, we describe the current state of research with respect to fault injection, covering both physical aspects and the algorithms needed to actually exploit an induced fault. In the following, we often refer to the device being analysed and attacked as the *Device Under Test* (DUT).

2.1. Side-Channel Analysis

As introduced in Chap. 1, side-channel analysis includes all techniques that allow for the analysis of a (cryptographic) algorithm by observing the behaviour of its physical implementation. Usually, the aim is to recover a cryptographic secret (i.e., a key), however, the methods may also be used to obtain other information, such as memory content.

In this thesis, we focus on *power analyis*, i.e., target DUTs where the side-channel information is recorded by (directly or indirectly) monitoring the power consumption of the device. The measured timeseries is usually referred to as *power trace* (or *trace*). More specifically, we address *Simple Power Analysis* (SPA) to *profile* the DUT and *Differential Power Analysis* (DPA) to recover the key. Other approaches such as *templates* or *timing attacks* are not explicitly covered, however, the framework described in Chap. 3 basically also supports such methods.

2.1.1. Simple Power Analysis

According to [MOP07], SPA attacks directly interpret single power traces to gain insight into the internal workings of a cryptographic device or to obtain the key. If the power consumption of an algorithm heavily depends on the key bits (due to different instructions being executed) as for instance in a straightforward implementation of a binary

exponentiation algorithm [KJJ99], the key can be recovered by observing the different patterns occuring when processing a zero or one, respectively.

On the other hand, SPA can be utilised to profile a device prior to performing other attacks. E.g., it is often mandatory to identify features in a trace that indicate the start of a cryptographic operation, allowing to narrow down the region that is subject to further analysis.

2.1.2. Differential Power Analysis

To exploit *statistical* dependencies of the trace on the processed data, Kocher et.al. proposed to execute a cryptographic operations several times with varying inputs, followed by a statistical test to reveal the correct *key candidate* [KJJ99]. If the side-channel information is obtained by measuring the EM emanation, the attack is called *Differential Electro-Magnetic Analysis* (DEMA) [Car05].

For differential attacks, each key candidate K_s, $0 \leq s < S$, where the number of candidates S should be small[1], is input to a *prediction function* $d(K_s, x_l)$, establishing a link between the l'th input data x_l and the expected power consumption for each key candidate K_s. Often, d predicts the power consumption of the output of an S-Box after the key addition, in many cases modelled based on the Hamming weight, i.e., the number of ones in a data word, or based on the Hamming distance, i.e., the amount of toggling bits in a data word. The recorded trace of length N is denoted as $\vec{t}_{x_l}(n)$.

The original test described by Kocher assumes that $d(K_s, x_l) \in \{0, 1\}$. Fix one key candidate K_s and group the traces according to the value of the prediction function into two sets $S_{K_s, d=0}$ and $S_{K_s, d=1}$. The means of the two sets are given as

$$\begin{aligned} \vec{m}_0(K_s, n) &= \frac{1}{|S_{K_s, d=0}|} \sum_{\vec{t} \in S_{K_s, d=0}} \vec{t}(n) \\ \vec{m}_1(K_s, n) &= \frac{1}{|S_{K_s, d=1}|} \sum_{\vec{t} \in S_{K_s, d=1}} \vec{t}(n) \end{aligned}$$

Then, consider the *difference of means*

$$\Delta_{DPA}(K_s, n) = \vec{m}_0(K_s, n) - \vec{m}_1(K_s, n) \tag{2.1}$$

By identifying K_s for which $|\Delta_{DPA}(K_s, n)|$ contains the peak with the highest amplitude, the correct key candidate can be revealed, given that enough traces have been

[1] This is always the case when attacking single S-Boxes with few in- and outputs

acquired and that there exists a link between the side-channel leakage and the prediction based on the input data.

Correlation Power Analysis

Correlation Power Analysis (CPA) [BCO04] generalises the original approach of DPA and enables the use of prediction functions with a real-valued output, i.e., $d(K_s, x_l) \in \mathbb{R}$. The method essentially relies on calculating the *normalised correlation coefficient* between the predicted and recorded values for one point in time n and a fixed key K_s, as given in Eq. 2.2.

$$\Delta_{CPA}(K_s, n) = \frac{\sum_{l=0}^{L-1} \left(\vec{t_{x_l}}(n) - m_{\vec{t}(n)}\right) \left(d(K_s, x_l) - m_{d(K_s)}\right)}{\sqrt{\sigma^2_{\vec{t}(n)} \sigma^2_{d(K_s)}}} \tag{2.2}$$

with $m_{\vec{t}(n)}$, $m_{d(K_s)}$ denoting the sample means and $\sigma^2_{\vec{t}(n)}$, $\sigma^2_{d(K_s)}$ the sample variances of the respective timeseries. Plotting $\Delta_{CPA}(K_s, n)$ for all n yields a curve indicating the correlation over time that features significant peaks, if K_s is the correct key guess, and has a random distribution otherwise. Thus, by iterating over all K_s and analysing the resulting $\Delta_{CPA}(K_s, 0) \ldots \Delta_{CPA}(K_s, N-1)$, the secret is recovered, again under the conditions mentioned for DPA.

2.2. Fault Injection

In contrast to side-channel analysis – which is a passive technique – fault injection involves active manipulation of a DUT to enable computations that eventually lead to the recovery of a cryptographic key. The initial research on the physical aspects of fault injection originates from investigations of semiconductor manufacturers to evaluate the reliability of their products [BP03]; the idea to attack cryptographic devices by means of inducing faults during the appropriate computations is relatively new [BDL97].

2.2.1. General Parameters of Fault Injection

Before detailing the diverse methods to inject faults in ICs, we identify general properties of faults in order to provide a model that helps to characterise the requirements for concrete attacks.

Permanence If a fault injection permanently alters the DUT, for instance, destroys a hardware part or overwrites the firmware, it is said to be *permanent*. Otherwise,

if the fault only affects the outcome of a limited number of computations, it is *non-permanent* or *transient*.

Precision of Time Position Subsequent attacks may require the fault to occur either at a *random* (indeterminate) position, within some *region* or at a *precisely determined* point in time.

Number of Affected Bits A fault is *single-bit* if it alters exactly one bit, or *multi-bit*, if it changes ≥ 2 bit, e.g., the state of a complete register.

Effect The induced modification can become manifest in a *bit flip*, i.e., logic values are inverted, a *fixed state*, i.e., logic values are tied to 0 or 1, or *inconsistent behaviour* of the DUT. In the latter case, the fault injection causes inconsistencies in the state of a device by affecting a distinct part of its digital control logic. A common example of this effect is the skipping of instructions on a microcontroller, e.g., due to the instruction pointer being incremented but the current instruction not being executed.

2.2.2. Types of Physical Faults in Integrated Circuits

There is a variety of ways to trigger faulty behaviour in ICs, differing (amongst others) in complexity, cost, effectiveness and the possible effects caused by the fault. In the following, we give a brief survey of methods that have been proposed in the literature.

Microprobing

One of the most direct yet complicated fault injection methods is to de-package the silicon die and contact a specific circuit path using *microprobes*. As detailed in [KK99], the attacker is then able to exactly monitor the waveforms present on this wire and can actively modify the value, for instance by short-circuiting it to ground.

Due to the immediate access to the DUT, virtually all types of faults can be injected. Moreover, the method allows for reverse engineering of the circuit. However, the needed equipment is expensive (in [KK99], the authors estimate a cost of 10000 - 100000 $) and requires considerable skill and experience to be handled efficiently. Additionally, the invasive nature of the attack makes it unusable in scenarios where permanent, obvious physical modification of the DUT is detectable, e.g., for identification or payment smartcards. For these reasons, we decided not to further detail on microprobing attacks in this thesis.

Temperature Variation

Since the characteristics of circuit elements vary with temperature, an *Integrated Circuit* (IC) only works correctly within the temperature range specified by the vendor. Thus,

cooling or heating the DUT and operating it outside of its maximum specifications can lead to faulty behaviour. [HCN+04] states that high or low temperature especially affects memory cells and can cause random modification of *Static Random Access Memory* (SRAM) cells or disable read/write operations of *Non-volatile Memory* (NVM), i.e., *Electrically Erasable Programmable Read-Only Memory* (EEPROM) or Flash. Generally, exact timing of the fault is complicated due to the limited thermal conductivity of the involved materials, e.g., the IC package or the die itself. Besides, most of the fault parameters mentioned in Sect. 2.2.1 are hard to control with this approach, limiting the possible application scenarios. In the current version of our setup, temperature variations can only be applied manually, i.e., using coolant spray or heating devices.

Optical Effects

By exposing the circuit to white or laser light, electron-hole pairs are created that can cause current flow at p-n junctions [Sch08, HCN+04] of semiconductors, resulting in changes of logic levels in the affected region of the IC, e.g., switch a transistor. By applying a mask or optical lenses to focus a small area, optical faults allow for precise targeting of certain parts of a circuit, down to the single-transistor level [WW05], with fine control over the fault effect.

Note that inducing optical faults is a semi-invasive attack, as the plastic packaging of the chip has to be opened mechanically or by etching, which is straightforward for standard IC packages, e.g., *Dual Inline Package* (DIP) or *Small-Outline Integrated Circuit* (SOIC), but can become infeasible for a common adversary in the case of highly sophisticated smartcards.

Variation of Power Supply

Temporarily increasing (*positive glitch*) or reducing (*negative glitch*) the supply voltage of an IC to a certain level is a well-established method to inject faults [GT04, FML+03], particularly with regard to the skipping or misinterpretation of processor instructions.

As the power is supplied via an external pin (for the case of most embedded device) or the surrounding EM field (for contactless (RFID) devices), the fault injection path is easily accessible, allowing for non-invasive attacks. However, at the same time, this single entry point can be disadvantegous from an attacker's point of view: Countermeasures such as monitoring or filtering the supply voltage before it enters the core of the circuit are relatively inexpensive, because they only need to be implemented for one isolated section of the IC.

Electro-Magnetic Pulses

Transients of the EM field cause induction of currents in conductors and can thereby change logic levels present on an IC. In contrast to power glitches, the fault injection is not performed over a single wire. Rather, the fault can affect any part of the DUT, making it harder to prevent and detect than variations of the supply voltage. On the other hand, the actual outcome is hard to predict and reproduce, since the location and extent of the fault depends on the exact position in and the form of the EM field, which in turn is subject to environmental conditions.

Variation of an External Clock

For devices with external oscillators, i.e., for which manipulations of the clock signal are feasible, slightly modifying the clock period for one or few (half-)cycles may lead to data corruption [KK99]. Due to different delays of distinct circuit paths, values that take longer to propagate (e.g., because they are transported over the *critical path*[2]) may not be handled correctly in the following clock cycle.

2.2.3. Methods for Fault Injection Attacks on Cryptographic Algorithms

In the following, we give an overview of the current research status for attacks on cryptographic systems that base on fault injection. We focus on the most important algorithms for *asymmetric* and *symmetric* cryptography, assessing their vulnerabilty to faults.

Since the presented attacks are initially theoretical and thus usually abstract the physical process of fault injection by specifying a fault model, the actual method for causing the fault is left open in this section. Rather, we try to systematise the great number of publications in this field by characterising:

1. The *targeted computation*, i.e., the part of the algorithm that is manipulated by means of fault injection,
2. the *fault model* following Sect. 2.2.1,
3. the *adversary model*, i.e., the type and number of cryptographical operations an attacker must be able to execute and
4. the *attack*, given as a pseudo-code algorithm in Appendix A.

[2]The critical path is the register-to-register path with the largest delay and thus limits the maximum clock frequency

Bellcore-Attack on RSA with Chinese Remainder Theorem Optimisation

The first fault injection attack on cryptographic algorithms was developed by Boneh et. al. at the Bellcore security and cryptography research group [BDL97]. The attack targets an implementation of RSA using the *Chinese Remainder Theorem* (CRT) optimisation that we briefly review in Alg. 1.

Algorithm 1 RSA Signature Creation with CRT Optimisation.

Require: Private system parameters:
Private primes $p, q \in \mathbb{P}$, private exponent $d \in \mathbb{Z}_{\varphi(pq)}$, CRT parameters: $d_p = d \mod p$, $d_q = d \mod q$, $c_p = q^{-1} \mod p$, $c_q = p^{-1} \mod q$

Require: Public system parameters:
Modulus $n = pq$, public exponent $e = d^{-1} \mod \varphi(n)$, $e \in \mathbb{Z}_{\varphi(n)}$

Require: Plaintext $x \in \mathbb{Z}_n$ to be signed

Ensure: $y = x^d \mod n$

1: $x_p \leftarrow x \mod p$
2: $x_q \leftarrow x \mod q$
3: $y_p \leftarrow x_p^{d_p} \mod p$
4: $y_q \leftarrow x_q^{d_q} \mod q$
5: $y \leftarrow [c_p q]\, y_p + [c_q p]\, y_q \mod n$
6: **return** y

In contrast to many approaches for other cryptographic algorithms (of which we present some in the following), the requirements with respect to the precision and the effect of the fault are very low – it may occur during one of the two exponentiations and affect the outcome arbitrarily, i.e., does not depend on the precise modification of certain bits. As such, it is a very powerful attack.

Targeted Computation	This attack targets one of the exponentiations in Alg. 1 (line 3 or 4) and recovers the secret exponent d
Fault Model	Injection of an arbitrary, non-permanent fault in a region belonging to either exponentiation
Adversary Model	The adversary is able to request the signature of a freely-chosen message x at least two times and receives the corresponding signature. He also knows the public system parameters
Attack	The attack steps are given in Alg. 6 in Appendix A

Lenstra-Attack on RSA with CRT Optimisation

Lenstra noted that the idea of the Bellcore-Attack can be further enhanced [Len96], no longer requiring both a faulty and a correct signature. Thus, the approach also works for implementations where multiple signatures on the same plaintext cannot be acquired, e.g., in randomised authentication protocols etc. Apart from that, the attack has similar requirements as the Bellcore method.

Targeted Computation	This attack targets one of the exponentiations in Alg. 1 (line 3 or 4) and recovers the secret exponent d
Fault Model	Injection of a arbitrary, non-permanent fault in a region belonging to either exponentiation
Adversary Model	The adversary is able to request one signature of a known message x and receives the corresponding signature. He also knows the public system parameters
Attack	The attack steps are given in Alg. 7 in Appendix A

Attack on RSA without CRT Optimisation

After the discovery of the susceptibility of CRT-optimised RSA, the basic idea was extended to several other public key systems, including a standard RSA signature generation. In [BDH$^+$97], a method for recovering the private RSA key d is proposed, based on a fault in the exponentiation algorithm. Note that, however, the prerequisites with regard to the actual fault parameters are very strict, as exactly one bit of the private exponent d must be altered during the algorithm run.

Targeted Computation	This attacks targets one bit in the binary representation of $d = (d_{l-1} \dots d_1\ d_0)_2$ with length l
Fault Model	Injection of a bit flip, non-permanent fault on exactly one random bit of d
Adversary Model	The adversary is able to request one signature of a freely-chosen message x and receives the corresponding signature. He also knows the public system parameters
Attack	The attack steps are given in Alg. 8 in Appendix A
Notes	The attack returns one bit of d. It can thus be used repeatedly with varying parameters to recover the complete key

Elliptic Curve Cryptography

Especially in the context of resource-constrained embedded systems, RSA is often infeasible due to the long operands (e.g., 1024 ... 2048 Bit) necessary to guarantee a certain security level . A popular alternative is *Elliptic Curve Cryptography* (ECC), which provides equivalent security at lower computational cost with, e.g., 160 Bit operands [Kra04]. However, as for RSA, several fault injection attacks have been proposed to recover the private key. In order to explain those, we first review the basics of ECC.

Given a field $\mathbb{F}$, the points of an non-singular *Elliptic Curve* (EC) E over $\mathbb{F}$ are the solutions (x, y) to

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

together with the identity point ∞, given that the coefficients a_1, a_2, a_3, a_4, $a_6 \in \mathbb{F}$ yield a non-zero discriminant (for details, cf. [CFA+06]). Note that in many practical applications, a simplified formula with $a_1 = a_2 = a_3 = 0$ and $a_4 = a$, $a_6 = b$ is used.

One can define point addition $P_1 + P_2$ with $P_1, P_2 \in E$ and, based on this, scalar multiplication $d \cdot P_1$ with $d \in \mathbb{Z}$ such that the points on E form a group. The addition formula is e.g. given in [CFA+06], but in the context of fault injection, it is sufficient to know that addition does not depend on a_6 in any coordinate representation.

If one considers points $P' = (x', y') \in \mathbb{F}^2$ that not necessarily lie on E but one still uses the addition formula with the coefficients for E, the resulting operations are called *pseudo-addition* and *pseudo-scalar multiplication* and denoted as $\oplus$ and $\otimes$ in this subsection. Using the coefficients of the original curve E, a new coefficient a'_6 is computed as:

$$a'_6 = y'^2 + a_1 x' y' + a_3 y' - x'^3 - a_2 x'^2 - a_4 x'$$

Then, the coefficients a_1, a_2, a_3, a_4, a'_6 define another non-singular elliptic curve E' , for which

$$d \otimes P' = d \cdot P'_{E'} \tag{2.3}$$

holds, with $d \in \mathbb{Z}$ and $P' = P'_{E'}$, $d \cdot P'_{E'}$ are points on E'. The important consequence of this fact is that the scalar multiplication starting with an invalid point on E is equivalent to a normal scalar multiplication on another curve E'.

Cryptosystems based on EC usually rely on the hardness of the *Discrete Logarithm* (DL) problem in the point group of E, i.e., given points $P, d \cdot P$, determining d is computationally costly. Equation 2.3 shows a way to map this problem on E to another curve E' which might have properties that allow for faster computation of the discrete logarithm, i.e., which is *cryptographically weak*.

In [BMM00], the authors describe two variants of a fault injection attack on the scalar point multiplication algorithm for ECC signatures, each time for a basic implementation and practical protocols, i.e., *EC ElGamal* and the *Elliptic Curve Digital Signature Algorithm* (EC DSA). Since the latter are virtually extensions of the already rather involved basic case, we focus on the idealised scenario, i.e., do not assume a concrete protocol, in order to explain the idea of the technique.

Fault Injection at the Start of the Multiplication

The first variant assumes a modification of the base point P before the scalar multiplication $d \cdot P$ is carried out. As mentioned above, the attack can be extended to work for real-world applications, in which complete points are usually never output.

Targeted Computation	This attacks targets the register holding the base point P that is input to the multiplication algorithm
Fault Model	Injection of a single-bit flip, non-permanent fault in the base point P before the multiplication
Adversary Model	The adversary is able to request the multiplication of a freely chosen point P with the private key d multiple times and receives the result $d \cdot P$
Attack	The attack steps are given in Alg. 9 in Appendix A

An interesting aspect is that the attack solely relies on a property of the input point. Thus, in systems which do not check the validity of P, the fault can be injected simply by choosing a P' not on E. Only if the implementation explicitly verifies the validity of the input point before performing the multiplication, a physical fault injection has to be applied to alter the point after the check.

Fault Injection during the Multiplication

Besides, the authors also sketch an algorithm which does not require the input to be faulty, but rather allows for inducing faults during the multiplication. It assumes the use of the binary double-and-add algorithm given in Alg. 2 for the scalar multiplication. For explanatory reasons, we also indexed the intermediate values Q and H. In real implementations, not all subsequent intermediates but only the current states are stored.

Algorithm 2 Right-to-left double-and-add algorithm.

Require: Elliptic curve E
Require: Point addition $+$ on E with identity point ∞
Require: Input point P on E
Require: Scalar $d \in \mathbb{Z}$ and its binary representation $(d_{n-1} \dots d_1\ d_0)_2$ with length n
Ensure: $Q_n = d \cdot P$

1: $H_0 \leftarrow P$
2: $Q_0 \leftarrow \infty$
3: **for** $i \leftarrow 0 \dots n-1$ **do**
4: **if** $d_i == 1$ **then**
5: $Q_{i+1} \leftarrow Q_i + H_i$
6: **else**
7: $Q_{i+1} \leftarrow Q_i$
8: **end if**
9: $H_{i+1} \leftarrow H_i + H_i$
10: **end for**
11: **return** Q_n

The number of faulty outputs required and the computational effort for Alg. 10 is substantially higher than Alg. 9, however, the timing requirements are relaxed. Still, both the realisation of the fault injection and the processing to recover the private exponent require considerable effort. With high probability, the recovery algorithm outputs only one candidate for a certain number of bits of the private key. To obtain the full key, the algorithm can then be used iteratively, with the steps belonging to the already known bits removed from the result of the multiplication, until the complete key has been determined.

Targeted Computation	This attack targets the register holding an intermediate result during the multiplication
Fault Model	Injection of a random, single-bit, non-permanent fault in the intermediate register Q of the multiplication algorithm 2 within a region of m successive iterations. The attack can also be adapted to work for the intermediate register H
Adversary Model	The adversary is able to request the multiplication of a freely chosen point P with the private key d multiple times and receives the result $d \cdot P$
Attack	Alg. 10 in Appendix A returns candidates for the most significant bits of d.

Differential Fault Analysis of the Data Encryption Standard

The methods to attack symmetric cryptographic systems are initially similar to those described for public key algorithms and usually exploit relations between faulty and correct outputs. For the *Data Encryption Standard* (DES) [FIPb], Shamir et. al. proposed an attack on the final round (cf. Alg. 3) [BS97].

Algorithm 3 Final round of DES.

Require: 64 bit DES state register $(L_{15},\, R_{15})$ after round 15

Require: 48 bit subkey K_{16}

Require: DES permutations $IP^{-1}(\cdot)$, $Exp(\cdot)$, $P(\cdot)$ and the S-Box layer $S(\cdot)$ formed by S-Boxes $S_1 \ldots S_8$

1: Round function: $f(R,\, K) = P(S(Exp(R) \oplus K))$ with $R \in \{0,\, 1\}^{32}$, $K \in \{0,\, 1\}^{48}$

2: $R_{16} \leftarrow L_{15} \oplus f(R_{15},\, K_{16})$

3: $L_{16} \leftarrow R_{15}$

4: Ciphertext: $y \leftarrow IP^{-1}(L_{16}, R_{16})$

5: **return** y

The attack requires a single-bit error in the input of one S-Box, causing ≥ 2 output bits to toggle. To recover the subkey for all S-Boxes, several plaintexts have to be sent, with the bit position of the fault slightly varied to affect all S-Boxes and consequently recover all subkey bits.

Targeted Computation	This attack targets the right half of the DES state register before the final round 16, summarised in Alg. 3. See [FIPb] for further details
. **Fault Model**	Injection of single-bit flip, non-permanent fault at the moment of the register update for R15
Adversary Model	The adversary is able to request the encryption of freely-chosen messages $x \approx 200$ times and receives the corresponding ciphertexts
Attack	The attack steps are given in Alg. 11 in Appendix A

Furthermore, it only gives 48 of the 56 DES key bits. Thus, either multiple rounds have to be targeted in succession, or the remaining bits are obtained by an exhaustive search.

Giraud's Differential Fault Analysis of the Advanced Encryption Standard

As it is the case with the preceeding standard DES, the *Advanced Encryption Standard* (AES) [FIPa] is susceptible to fault injection. In [Gir03], an attack on the final round (cf. Alg. 4) is outlined. Here, all descriptions are given for the 128 bit variant AES-128 for simplicity, but they can also be extended to AES-192 and AES-256.

Algorithm 4 Final round of AES.

Require: 128 bit AES state register M_9 after round 9

Require: 128 bit subkey K_{10}

Require: AES operations $SubBytes(\cdot)$ and $ShiftRows(\cdot)$

1: Ciphertext: $C \leftarrow ShiftRows(SubBytes(M_9)) \oplus K_{10}$

2: **return** C

Targeted Computation	This attack targets the AES state register before the final round 10, summarised in Alg. 4. See [FIPa] for further details
Fault Model	Injection of a single-bit flip, non-permanent fault in the state M_9 after round 9
Adversary Model	The adversary is able to request the encryption of freely-chosen messages x at least $50 + 1$ times and receives the corresponding ciphertexts
Attack	The attack steps are given in Alg. 12 in Appendix A

Blömer-Seifert Fault Attack on AES

AES includes a pre-whitening step (Alg. 5), i.e., an initial key addition, which can be aimed at with fault injection [BS03]. Fig. 2.1 displays the structure of the first round, including the pre-whitening. Again, all descriptions are for the AES-128, but the idea is directly applicable to the variants with longer operands.

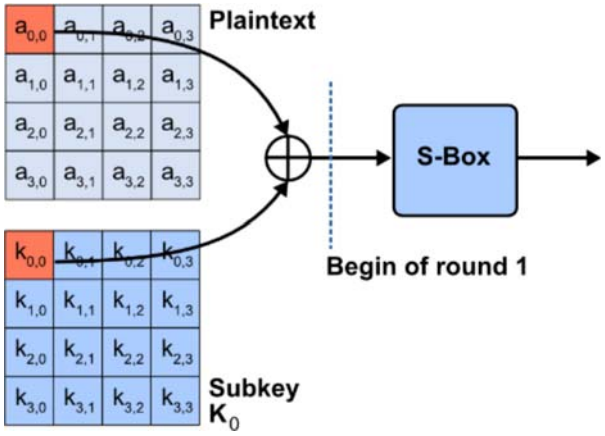


Figure 2.1.: First subkey addition and S-Box of AES for first plaintext byte (based on figure in [Wik09]).

Algorithm 5 Pre-whitenining step of AES.

Require: 128 bit AES plaintext P

Require: 128 bit subkey K_0

1: Input for first round M_0: $M_0 \leftarrow P \oplus K_0$

2: **return** M_0

The assumed fault unconditionally sets one bit of the first round input M_0 after the key addition to zero (or, equivalently, one). Then, the outcome of the encryption is observed and compared to the correct ciphertext. If the results match, the fault has not changed the state of the targeted bit in M_0, i.e., the value of the bit is zero (or one, respectively), and, since the plaintext bit is known, the corresponding key bit can be computed. By scanning through all bits, the complete key is recovered.

Targeted Computation	This attack targets the AES state register after the pre-whitenining step, summarised in Alg. 5. See [FIPa] for further details
Fault Model	Injection of a fixed-value, single-bit, non-permanent fault in the state M_0 before round 1. The fault affects exactly one bit which is unconditionally set to zero
Adversary Model	The adversary is able to request the encryption of freely-chosen messages x at least 129 times and receives the corresponding ciphertexts
Attack	The attack steps are given in Alg. 13 in Appendix A

The applied technique is particularly interesting because the underlying idea is rather general and can be applied to a wide variety of other cryptographic algorithms where (a part of) the key is directly added to the plaintext. Yet, the fault model – setting a bit to a fixed value – is quite specific, requiring further research on the practical feasibility of such attacks.

Bypassing Vendor-specific Protections

Aside from breaking crypographic algorithms, fault injection techniques have been reported to be capable of overriding vendor-specific security features such as read- and write-lock bits of microcontrollers that — once set during manufacturing — prohibit any further access to the instruction memory. Today, embedded software is considered a valuable intellectual property, which an adversary, e.g., a competitor, must not be able to reverse-engineer. Additionally, knowing implementation details can significantly simplify a subsequent side-channel analysis or fault injection. For many common microcontrollers, methods exist to bypass code protection by inducing a fault while issueing the programming command [Sko01, Bre09].

A more sophisticated method to keep the embedded code secret is found in FPGAs, e.g., Virtex products [Xil07], manufactured by Xilinx. They allow for complete encryption of the bitstream used to program the FPGA with a symmetric cipher. In this case,

fault injection (as well as side-channel analysis) might be used to attack the implementation of the respective cipher to recover the secret key and subsequently read out the FPGA configuration.

2.3. Combined Attacks

Recently, a promising combination of fault injection and side-channel analysis has been proposed, termed *Passive Active Combined Attack* (PACA) [AVFM07]. The basic idea is to attack implementations that are protected against both active and passive attacks separately, but do not take the effects of applying both methods at the same time into account.

To illustrate the impact of this type of attack, consider a *masked* implementation of a cryptographic algorithm, i.e., the statistical dependency of the power consumption on the data is eliminated by incorporating a random value called a *mask* into the internal computations [MOP07]. If an adversary can induce a fault during the generation or loading of the mask, setting it to a constant or weak value (e.g., completely zero), subsequent side-channel analysis is enabled, as the countermeasures depends on the randomness of the mask that no longer holds.

In [AVFM07], the authors target an implementation of RSA [CMCJ04] that protects against both SPA and DPA and additionally incorporates message randomisation. By skipping the initialisation of a mask register and leaving it set to zero, the outcome of a multiplication is no longer randomised, resulting in a strong dependence on the private key bits which can be spotted in the power trace. Thus, a straightforward SPA can be mounted.

Both the possibilities of combined attacks and feasible countermeasures against PACA are currently investigated by the scientific community. As already the first proposed attacks pose a severe threat to most present implementations and are difficult to protect against, further significant results can be expected to be published in the near future. The framework presented in this thesis is thus designed to support combined attacks to be able to quickly obtain practical results for upcoming approaches.

3. A Framework for Side-Channel Measurements and Fault Injection Attacks

In this chapter, the framework for performing side-channel and fault injection attacks (and possibly combinations of both) developed in this thesis is presented. We describe the important building blocks and give details on the actual realisation and the *Application Programming Interface* (API) used on the *controlling PC* to communicate with the respective part.

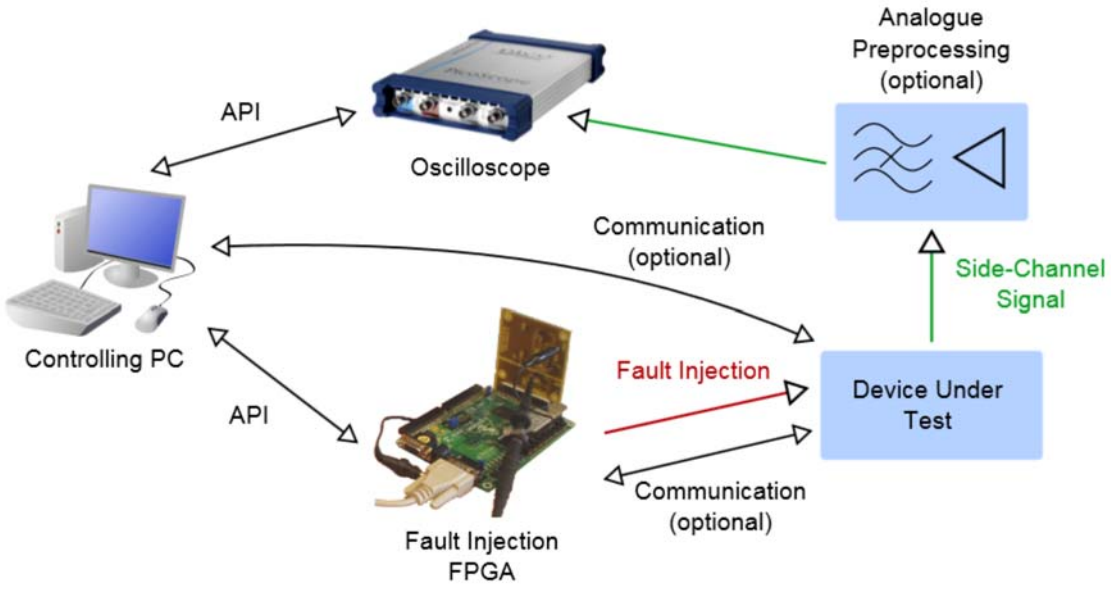


Figure 3.1.: Overall system structure for combined side-channel analysis and fault injection.

Figure 3.1 gives an overview over the general system structure: the controlling PC connects to an oscilloscope to record analogue signals, communicates with the DUT and configures a *Field Programmable Gate Array* (FPGA) that performs the fault injection and is optionally able to control the DUT as well. In addition, there are analogue

preprocessing circuits in order to improve the signal quality prior to oscilloscope measurements. Note that in addition to the description here, a source-level documentation using *Doxygen* is available, providing detailed information for all framework components, as summarised in Appendix B. The complete documentation in HTML format is provided together with the source code of the framework.

3.1. Development Toolchain

The development of the software for the controlling PC is done in C++, using the *GNU Compiler Collection* (gcc) [GCC09] under *Cygwin*, a Linux-like environment for Windows [Red09]. Hence, porting to other operating systems, e.g., Linux, is possible, as long as the vendor-specific APIs (e.g., for the oscilloscope) are available for that platform. All applications are compiled using *Makefiles*: in order to build the executable, running `make` in the respective directory performs the necessary steps, provided that the library dependencies (`fftw3`, `ncurses`, vendor-specific APIs) are met.

The *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) code is created, synthesised and mapped for the used FPGA with the free *Xilinx ISE Design Suite WebPACK* [Xil09a]. The configuration bitstream is written to the FPGA using a *Joint Test Action Group* (JTAG) cable connected to the parallel port of the development PC.

3.2. Configuration Files

Many components of the framework are controlled by *configuration files*, so that the parameters of the respective part are kept separate from the program code. The basic idea is to have a folder for each series of measurements that contains – in addition to the recorded waveforms – all configuration data for the recording and subsequent analysis. This approach improves the reproducibility and facilitates the documentation and evaluation of measurements, and allows for, e.g. automatic batch processing of the results.

The configuration files use a key-value pair syntax and are grouped in sections belonging to several sub-modules. We detail the available settings for each module in the according section. The following example fragment illustrates the syntax of a configuration file:

```
# ...
[scope]
# Which scope is used: picoscope (default), adc8, adc16, adc32
```

```
type = picoscope
# Sample rate in Hz
sample_rate = 1000000000
# ...
```

Thus, a section is indicated by specifying the name in square brackets, followed by a number of lines with key-value pairs (separated by `=`). A `#` indicates a comment, i.e., all following characters on this line are ignored. The values of specific keys are read in using the open source *iniParser* library [Dev08], which provides functions such as `iniparser_getstring()`, `iniparser_getint()` or `iniparser_getdouble()` to obtain values of the respective type.

For better usability, the framework provides the classes `measurement_config` and `processing_config` which parse the configuration files and wrap the access to the specific settings. For this purpose, those classes contain several sub-objects matching the sections of the configuration file, e.g., `scope()` for accessing the oscilloscope configuration. In turn, these sub-objects then allow to actually read the settings with `get...()` methods.

3.3. Communication Modules

Though it is possible to conduct most implementation attacks using commercially available equipment, custom devices for communicating with the DUT are beneficial. Commercial products often rely on proprietary ICs which do not give full control over the exchanged data. E.g., a reader device generates the nonces for a challenge-response-protocol using a built-in *Random Number Generator* (RNG), computes parity bits and checksums, encrypts and sends data, without that an adversary can directly influence the process.

Thus, in the following we present customised readers for possible DUTs, tailored to the requirements of implementation attacks, in order to gain complete control over the communication, i.e., send arbitrary commands or repeatedly the same chosen plaintext, intentionally transmit wrong checksums and — crucial in the context of implementation attacks — have full control over the timing and generate precise trigger signals. Note that it is often sufficient to implement only some part of the protocol, up to the point at which the DUT performs the respective cryptographic operation.

3.3.1. ISO 7816 Smartcard Reader

For communicating with ISO 7816 smartcards, the operating system driver routines in `winscard.dll` are wrapped to simplify the use of the rather tedious low-level smartcard

API calls. These methods (implemented in `aes_smartcard.(h/c)`) are specific for an example smartcard described in Sect. 4.1.1, however, can also serve as a template for other similar DUTs.

aes_smartcard_open() Opens the connection to the smartcard and initialises the communication protocol.

aes_smartcard_close() Closes the connection to the smartcard.

aes_smartcard_setkey() Sets a new key for the AES encryption.

aes_smartcard_send_challenge() Sends a challenge to the smartcard that is encrypted with AES and returns the corresponding ciphertext.

When programming the interface for other smartcards based on this implementation, in most cases, the methods for opening and closing the connection can be left unchanged. Only the functions directly sending an *Application Protocol Data Unit Command* (ADPU), i.e., commands specific for the respective smartcard, have to be adapted accordingly. Usually, this involves the use of `send_apdu()`, which is `static`, that is, only available within the API source file.

3.3.2. Contactless Smartcard Reader

In [Kas06], a flexible customised circuit for realising ISO 14443A-compliant [iso01a, iso01b] 13.56 MHz RFID reader functionality and tag emulation is presented. Commands are sent via a USB interface from the controlling PC. In the context of our framework, we currently only employ the functions needed to communicate to contactless smartcards. Accordingly, we implement the base class `rfid_device` that provides all basic methods required for the standard RFID protocol features. Subsequently, this class can be inherited and used to provide the application-specific commands for authentication and encryption for a concrete PICC. For this purpose, the following functions are of special importance.

open() Opens the connection to the PCD.

close() Closes the connection to the PCD.

send_frame() Sends a plain ISO 14443A frame.

send_crc_frame() Sends an ISO 14443A frame, appending a checksum according to the standard.

read_result() Returns the answer sent by the RFID tag in response to a command issued by the PCD.

check_remove_parity() Checks and removes the parity bits inserted into a PICC response.

Configuration File

For the configuration of the RFID reader and the protocol necessary to communicate to contactless smartcards, a respective section of the configuration file may be used. First, the serial port of the controlling PC for connecting to the reader is specified, followed by the configuration which type of challenges are to be sent during the side-channel measurement phase.

`diff_means` indicates a mode in which only two challenges are sent alternately, in order to be able to isolate the points in time where the side-channel information is leaked during the profiling phase. In `full_dpa` mode, the challenge cycles through the 2^6 possible input values for one isolated S-Box of the DES, as proposed in [Osw08]. The targeted S-Box for those two modes is specified by `sbox`. In contrast, `full_dpa_rand` selects the transmitted challenges uniformly distributed at random. In all modes, the sent values and additionally the responses of the PICC during the protocol execution are stored in standard text files, with 8 bytes in hexadecimal encoding per line.

```
[reader]
# COM port of reader
port = "/dev/com3"

# diff_means, full_dpa, auth, full_dpa_rand
mode = full_dpa_rand

# sbox index 1 - 8 for diff_means and full_dpa
sbox = 1
```

3.3.3. Arbitrary Parallel/Serial Communication

Embedded systems and cryptographic devices that do not feature an RFID- or ISO 7816-based interface usually employ a serial or parallel protocol to communicate with their environment. In the context of implementation attacks, example targets may be FPGAs that are configured with an encrypted bitstream or cryptographically protected USB dongles. Therefore, we support a variety of corresponding protocols, either by the controlling PC if the timing is not crucial (e.g., USB, RS-232, or parallel port) or by means of the fault injection FPGA platform, if precise timing is required (e.g., *Serial Peripheral Interface* (SPI) and general purpose I/O pins). If necessary in future applications, further methods can easily be added thanks to the modular nature of our setup.

3.4. Measurement Modules

To record and evaluate side-channel information, modules for measuring and filtering the analogue signal, controlling the oscilloscope, and finally processing the recorded traces are available. In the following section, we describe these parts of the framework.

3.4.1. Oscilloscope

A *Digital Storage Oscilloscope* (DSO) is used to record analogue waveforms to develop, prepare and to actually conduct side-channel measurements. In its current version, the frameworks employs the *Picoscope 5204*, a dual-channel DSO controllable from a PC via its USB 2.0 interface [Pic08]. It features a maximum sample rate of 1 GHz, 8 bit vertical resolution at an accuracy of $\pm 3\,\%$ and 128 MSamples waveform memory. The input bandwidth is 250 MHz, with a minimum input range of $\pm 100\,\mathrm{mV}$. Additionally, the device offers a separate external trigger input and a built-in arbitrary signal generator.

Note that the API is kept as abstract as possible so that different oscilloscopes can be substituted, e.g., when higher precision or bandwidth is needed. We address the general API and the Picoscope-specific parts in this section.

Configuration File

In order to be able to adjust parameters of the oscilloscope without changing and recompiling source code, most features can be controlled via the configuration files mentioned above. In the following, we present an example step-by-step. The first section specifies global settings, i.e., the type of the oscilloscope (for which currently only `picoscope` provides full functionality) and the sample rate in Hz.

```
[scope]
# Which scope is used: picoscope (default), adc8, adc16, adc32
type = picoscope
# Sample rate in Hz
sample_rate = 500000000
```

The following parts already depend on certain characteristics of the concrete oscilloscope, as e.g., the number of available channel varies depending on the respective product. For the Picoscope 5204 with two analogue inputs, the configuration includes the activation of each channel, the range in mV, the coupling[1], the use as a trigger

[1] AC coupling implies a highpass filter which removes a DC offset before analogue-digital conversion

source, the trigger mode[2], threshold and hysteresis in mV, and finally the prefix which is used to name the files containing the waveform data recorded on this channel.

```
# Settings for first scope channel
[channel_a]
# true, false (default)
enable = true
# in mV (defaults to 100 mV)
range = 100
# Coupling: ac (default), dc
coupling = ac
# Enable as trigger source: true, false (default)
trigger_enable = false
# Filename prefix (output format: prefix trace_number.dat)
file_prefix = trace_channel_a

# Settings for second scope channel
[channel_b]
# true, false
enable = true
# in mV
range = 10000
# coupling: ac (default), dc
coupling = dc
# enable as trigger source: true, false
trigger_enable = true
# level, window
trigger_mode = level
# Threshold for level trigger, lower threshold for window trigger
# in mV, defaults to range/2
threshold_lower = 1500
# Ignored for level trigger, upper threshold for window trigger in mV
threshold_upper = 1500
# Hysteris for trigger in mV (defaults to 50 mV)
hysteresis = 150
# Filename prefix (output format: prefix trace_number.dat)
file_prefix = trace_channel_b
```

In the example file, only the second channel B is used for triggering the start of the recording. Additionally, the range and trigger configuration for the external trigger input

[2] A trigger event may either occur when a certain voltage level is reached or when the voltage is within some interval or window

has to be specified, however, note that this channel cannot be used to actually digitise the analogue signal.

```
# Settings for external scope channel
[channel_ext]
# true, false
enable = false
# Enable as trigger source: true, false
trigger_enable = false
```

Next, settings with respect to the amount of data (in ns) gathered before a trigger and the delay of the start of the recording with respect to the trigger event (in ns) are made. Additionally, it is possible to define an automatic trigger after a certain period of time (in ms) without an event. The conditions on which a trigger event shall occur are defined following the approach of the Picoscope API.

It allows for rather flexible combination of the trigger states of the individual input channels to form the global trigger signal which in turn starts the actual recording. For each channel, a direction (rising and/or falling edge, above or below a level) and the condition which must be met to generate a trigger event are specified[3]. These conditions are combined with a logical AND, i.e., must all be met simultaneously for a trigger event to be caused. In the example, only channel B is used, reacting to a rising edge.

```
[scope_trigger]
# Number of samples before trigger in ns
pretrigger_samples = 5000
# Delay after trigger before recording in ns
trigger_delay = 0
# Automatic trigger after specific period, 0 to disable, in msec
auto_trigger = 2000

# Mode for channels: true, false, dont_care (default)
channel_a = dont_care
channel_b = true
channel_ext = dont_care

# Directions for channels:
# none (default), rising, falling, rising_falling, above, below
direction_a = none
direction_b = rising
direction_ext = none
```

[3] `true` indicates that the direction event has, `false` that it has not occurred; `dont_care` ignores the trigger state of the input channel

A special feature of the Picoscope 5204 is the pulse width qualifier, which can serve as a trigger source to detect certain characteristics with respect to pulse width in an input waveform. The example configuration assumes a negative pulse, i.e., a transition $1 \rightarrow 0 \rightarrow 1$, with a width of less than $1.4\,\mu$s. Again, the signal for which the pulse conditions are checked is formed by a logical AND of the individual channel states. Note that if this feature is used, the channels set to `true` here must also be configured in the above part, accordingly. Besides, the above setting `direction_b = rising` has to be changed to `none` so that a simple rising edge does not generate a trigger event, but only the pulse of the desired width.

```
[scope_pulse_width_trigger]
# Enable pulse width qualifier trigger: true, false
enable = true

# Mode for channels: true, false, dont_care (default)
channel_a = dont_care
channel_b = true
channel_ext = dont_care

# Direction of pulse: rising (positive pulse), falling (negative pulse)
direction = falling
# Pulse width recognition type:
# less_than, greater_than, in_range, out_of_range
mode = less_than
# Lower limit for pulse width recognition in ns
# (used for all modes)
lower = 1400
# Upper limit for pulse width recognition in ns
# (used for range modes only)
upper = 0
```

API

When controlling the oscilloscope directly from own programs and not using the framework applications that make use of the configuration files, an object-oriented interface can be used, encapsulating the details of a concrete oscilloscope. This simplifies the change to a different product – at least up to a certain extent, as the variety of vendor-specific features renders the creation of a completely device-independent API virtually impossible.

The abstract base class `scope` defines the common interface for controlling and retrieving waveform data from an oscilloscope. Any concrete implementation, e.g., for

the Picoscope, has to inherit this base class and provide code for the methods we briefly describe in the following. Refer to the Doxygen documentation for details like parameter types, return values and implementation-specific aspects.

connect() Opens the connection to the oscilloscope.

disconnect() Closes the connection to the oscilloscope, stopping any active recording.

configure() Apply the settings passed to this method, wrapped in an object of type `measurement_config`.

arm() Arm the oscilloscope, i.e., wait for trigger events.

disarm() Disarm the oscilloscope, i.e., stop reacting to trigger events.

fetchData() Retrieve the current waveform data for a certain channel in mV.

getRawDataWidth() Get the width of the internal data buffer in byte.

fetchRawData() Retrieve the current waveform data for a certain channel as raw integer values, where the width of one value can be otained by calling `getRawDataWidth()`.

We provide a concrete implementation of this interface for the Picoscope 5204, encapsulating the C-API supplied by the vendor [Pic07]. Basically, the process to configure and start the recording of a waveform consists of a few function calls. Note that, however, in the following example, parts like error checking have been omitted for better readability.

```
// load configuration file
measurement_config cfg("/path/to/config_file.ini");

// create oscilloscope instance
scope* s = new picoscope();

// open and configure
s->connect();
s->configure(cfg);

// wait for trigger event
s->arm();

while(!s->isDataAvailable()) {
::usleep(1000);
}

// retrieve waveform for channel A
timeseries_t data;
```

```
s->fetchData("a", data);

// ... process data ...

// close connection
s->disconnect();
```

3.4.2. Analogue Pre-Processing

Frequently, the quality of the signal carrying the side-channel information can be improved before it is digitised by the oscilloscope. Using specific measurement equipment, analogue filter circuits and pre-amplifiers is of particular importance when analysing the EM emanation of, e.g., an RFID tag.

Amplifier

Before addressing the equipment for specific attack scenarios, we note that it is often advantageous to amplify the typically small amplitude of the signal in the analogue domain to reduce the *Signal-to-Noise Ratio* (SNR). The commercially available amplifier PA303 made by Langer EMV features an almost constant gain of 30 dB (i.e., multiplication by ≈ 31.6) up to 3 GHz. The amplifier was designed for a characteristic impedance of 50 Ω, so appropriate termination may be required for high-impedance scope inputs. Besides, its output amplitude is, according to the vendor, limited to ≈ 1 V. Above this level, non-linear clipping effects occur. Thus, the input amplitude must not exceed ≈ 31 mV.

ISO 7816 Smartcards

For side-channel analysis of smartcards according to ISO 7816 [ISO04], we constructed an adaptor that fits into any commercial smartcard reader. On the other side of the adaptor, a socket for any ISO 7816 card is available. The data and power lines are tapped and rewired, so that the signals can be relayed from and to a standard reader, e.g., for monitoring communication protocols. The power pin allows to connect an external stable power supply (or, for fault injection, our module presented in Sect. 3.5.2), with a resistor inserted in series to the ground pin of the smartcard for recording power traces.

Contactless Smartcards

When conducting a DEMA, the EM emanation is captured by means of near-field probes [Lan] manufactured by Langer EMV[4]. The probes are connected via standard

[4]The probes were originally designed for electromagnetic compatibility (EMC) tests

BNC cables with a characteristic impedance of 50 Ω. For most measurements, the RF-U 5-2 probe is used, because it is suited to capture the near H-field which is proportional to the current flow on the IC.

In our previous work [Osw08], we proposed several filter circuits for reducing the influence of the sinusoidal reader field to facilitate DEMA on contactless smartcards. Our approach bases on subtracting a clean reference sine signal, e.g., tapped at the reader crystal oscillator, from the measured EM trace. To equalise the amplitudes and correct for the inevitable phase shift caused by the *Radio Frequency* (RF) interface of the reader, appropriate blocks are included in the circuit. Fig. 3.2 demonstrates the principle of this idea, for details, cf. [Osw08].

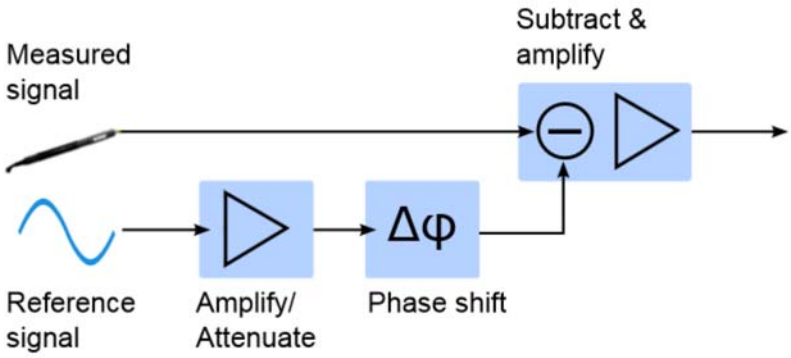


Figure 3.2.: Block diagram for reduction of RFID reader field influence.

3.4.3. Framework Class for Measurement Applications

The class `measurement_app` is designed to simplify the development of applications that perform typical side-channel measurements and fault injection attacks. Hence, the class encapsulates as much commonly required functionality as possible, while maintaining full flexibility, i.e., allows the user to modify these abstracted tasks if necessary. We follow a common approach employed in object-oriented frameworks and develop a base class with methods that are overwritten by inheriting classes to define the application-specific behaviour.

Fig. 3.3 depicts the control flow of a standard measurement application, with the methods highlighted that are commonly overwritten to customise the program for a concrete DUT. These methods mainly deal with establishing the connection (`initDUT()`, `cleanupDUT()`), send and update the challenge for the DUT (`sendChallenge()` and `nextTrace()`) and optionally process the recorded waveform (`processTrace()`). Note that fault injection is integrated with the measurement process, allowing for combined

and possibly adaptive attacks, i.e., choose the next challenge or discard waveforms based on the result of a prior fault injection.

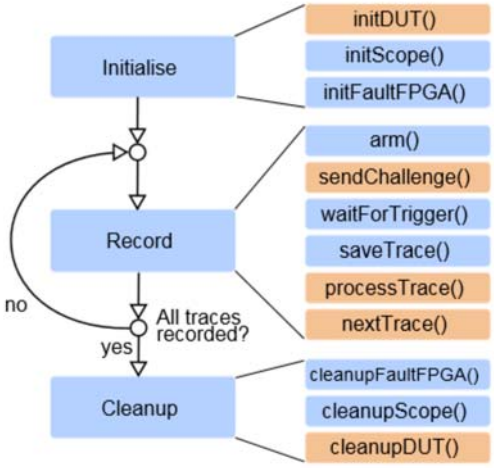


Figure 3.3.: Program flow of measurement framework application.

For more details on the usage of the framework class, please refer to the Doxygen documentation and the provided examples. These include a measurement application for a real-world smartcard, a combined fault injection and waveform recording program and demonstration code for further customising the framework for special requirements that are not covered by the basic structure.

Configuration File

The recording of waveforms can be customised with regard to the maximum number of trace files that are stored per directory. This is often useful as many filesystems respond significantly slower if a folder contains too many files. Besides, the total number of recordings and the length (in ns) of each single trace are set by the corresponding lines in the respective configuration section.

```
[record]
# Max. number of files in one dir
max_files_per_dir = 1000

# Number of traces to record
trace_count = 500

# Length of trace in ns
sample_count = 2300000
```

3.4.4. Framework Class for Evaluation and Processing Applications

Analogous to the framework facilities for capturing side-channel information and controlling the induction of faults, the separate class `processing_app` copes with the task of processing and evaluating the gathered waveforms, i.e., perform the actual analysis. The idea is similar to `measurement_app`, that is, certain methods are overwritten to realise the desired functionality, while repetitive procedures are provided by the base class. This includes the loading of the trace data files, digital pre-processing, alignment in time and in general the handling of configuration information. The program flow is outlined in Fig. 3.4, marking the methods commonly adapted with respect to the requirements of the specific problem.

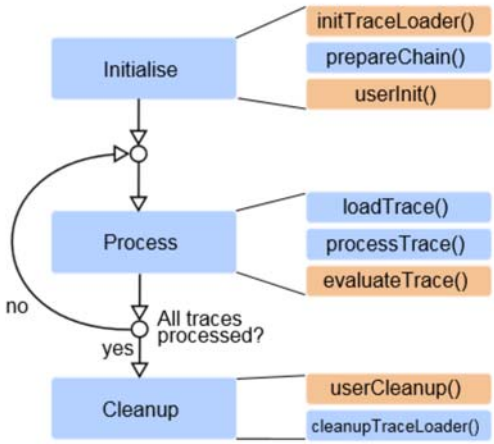


Figure 3.4.: Program flow of measurement framework application.

Classes inheriting `trace_loader` are employed for reading the waveform file format. When not specifying a custom loader, `default_trace_loader` is used, supporting the straightforward binary format[5] output by `measurement_app`.

During the evaluation process, each waveform is digitally pre-processed by the *processing chain*, passing the signal through a sequence of filtering blocks. The chain is set up in the configuration file and currently provides blocks for extracting parts of a timeseries, rectification, digital high- and lowpass filters, pattern matching functionality for alignment of power traces in time and the computation of the *power spectrum*, i.e., the squared magnitude of the Fourier transform [KK06].

[5]Essentially, these files contain the digitised signal samples as signed 16-bit integers in the case of the Picoscope 5204

The evaluation of the pre-processed signal depends on the concrete application and has thus been provided by the user of the framework. Examples include the extraction of local minima and maxima from a trace to reduce the amount of data before further evaluation, or the implementation of a CPA on a 3DES encryption engine.

Configuration File

We exemplify the configuration file for conducting a CPA on a 3DES engine. The first section references the configuration file from which certain settings are needed for the processing, e.g., the sample rate, the number of traces etc. Subsequently, parameters of the CPA, that is, more specifically, of the prediction function (predicted intermediate value, number of bits), the power model, and the location for the result files are set.

```
[common]
# Name of the recording settings file
settings_file = settings.ini

[3des_dpa]
# Name of challenge file
challenge_file = challenges.txt
# Output for correlation data
output_dir = /correlation_hump1_8MHz/
# Known-key for triple DES
key = 3b 38 98 37 15 20 f7 5e 92 2f b5 10 c7 1f 43 6e

# DPA oracle to use:
# known: Intermediate results before and after each DES are known
# known_first_des: All intermediate in first DES iteration are known
# dpa_first_round: CPA on first round of first DES
# dpa_des2_first_round: CPA on first round of second DES
oracle = dpa_first_round
# Power Model: weight, distance
power_model = distance
# For weight: For distance: 1st intermediate.
round1 = 0
# 2nd intermediate
round2 = 1
# Number of S-Box bits for CPA (1-4)
bits = 4
# Number of attacked S-Boxes for dpa_first_round
sbox_cnt = 8
```

The setup of the processing chain is done by specifying the respective blocks consecutively, where the section name is arbitrary – but must be unique within the chain – while the type is determined by the first line of each block. As an example, we use a processing chain useful for analysing RFID tags, which will be applied in Sect. 4.1.2 for a practical key-recovery attack on a contactless smartcard. Fig. 3.5 displays a graphical representation of the signal flow for this processing chain.

```
[cut]
type = cut_trace
# Where to start in trace (in ns)
begin = 18000
# How many samples to load (in ns)
length = 35000

[rectify]
type = rectifier

# demodulation lowpass
[lowpass]
type = bandpass
# Highpass cutoff, i.e. left corner frequency in Hz
cutoff_left = 0
# Lowpass cutoff, i.e. right corner frequency in Hz
cutoff_right = 8000000
# FIR filter order
order = 800

# DC block highpass
[highpass]
type = iir_highpass
# Highpass cutoff frequency in Hz
cutoff = 50000

[align]
type = alignment
# Number of reference trace
ref_trace = 2
# After which step of chain to get pattern
pattern_after = highpass
# Begin of pattern (in ns)
begin = 19370
# Length of pattern (in ns)
```

```
length = 600
# Offset for pattern search (in ns)
search_offset = 4000

# Disable automatic pattern search for databus
auto_align_databus = false

[cut2]
type = cut_trace
begin = 19000
length = 3500
```

Note that this example only employs a subset of the available blocks. We do not describe all parameters for all possible functions here, but rather present one particular example to emphasize the flexibility of the chain concept. It is easily extended with new functionality as needed and separates the settings for the pre-processing from the application logic, facilitating the re-use of verified primitives and – for experiments that are inevitable when attacking real-word DUTs – the batch processing with different settings.

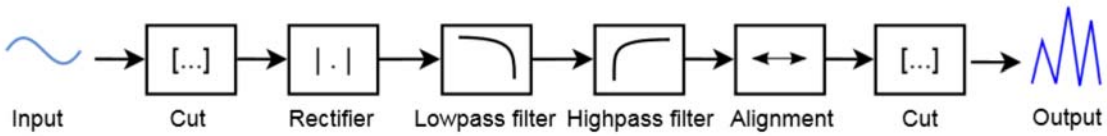


Figure 3.5.: Graphical representation of an example processing chain.

3.5. FPGA-based Module for Fault Injection

As summarised in Sect. 2.2.1, many different approaches can be utilised to inject faults in integrated circuits. In order to unify the application of this methods, we propose an FPGA-based control board which is extended with *fault modules* which realise the actual physical effect. The FPGA provides an interface to the controlling PC via an RS232 serial port, controls fault parameters (e.g., position in time, duration etc.) and is able to communicate with the DUT if required.

The use of an FPGA has certain advantages compared to a microcontroller-based solution, particularly with regard to precise timing of control signals at high clock frequencies. To minimise the design time, we use a commercial Xilinx Spartan-3 board [Xil08c] as a basis.

The system clock frequency is defined to 100 MHz, generated from the 50 MHz on-board oscillator using a Spartan-3 *Digital Clock Manager* (DCM). At 100 MHz, the FPGA runs significantly faster than most of the considered embedded systems (which usually are clocked at between ≈ 32 kHz and 20 MHz, cf. [Cor05]), thus enabling the injection of faults at multiple points during one clock cycle of the DUT. Note that at higher frequencies, it becomes increasingly difficult to select and apply the involved analogue (and digital) components appropriately. If future applications require higher frequencies, it is still possible to run the respective parts of the FPGA logic in a different clock domain, or even switch to a more powerful FPGA.

3.5.1. Overall System Structure

To simplify the implementation of complex control logic, the design is built around a general 8-bit microcontroller softcore (Xilinx PicoBlaze, cf. [Xil09b]) which is internally connected to several application-specific modules, as depicted in Fig. 3.6:

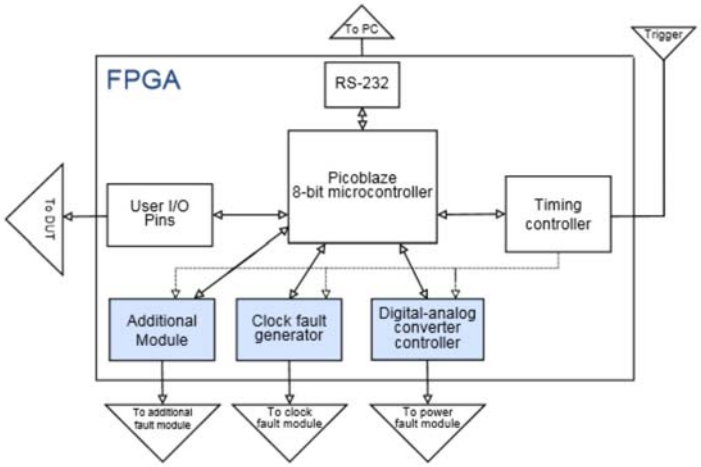


Figure 3.6.: FPGA structure overview.

Control Logic

The PicoBlaze softcore is a *Reduced Instruction Set Computing* (RISC) microcontroller, programmable using a simple assembler language [Xil08a]. It has low resource requirements (96 slices + 1 block RAM on a Spartan 3 FPGA), is supplied as VHDL source and is well suited for implementing non timing-critical control and interface logic.

All timing-critical operations that have to respond to external inputs instantly, and require guaranteed timing behaviour, are moved into the application specific blocks. The central module in this respect is the *timing controller*, which is responsible for

starting previously configured faults with precise timing parameters. The purpose of the microcontroller is to provide a unified and extensible interface between the controlling PC and the actual fault injection modules. The complete embedded code (firmware) of the PicoBlaze is contained within the file `control.psm` and can be converted to a VHDL instruction ROM using the assembler `kcpsm3.exe`.

Interface to Controlling PC

Due to its simplicity and the option for galvanic isolation if required, an RS232 *Universal Asynchronous Receiver Transmitter* (UART) is used to control the fault injection FPGA from the central PC. For this purpose, Xilinx includes the VHDL module `uart` with the PicoBlaze package, which is connected to the external bus of the softcore and can be accessed to receive and transmit bytes with few assembler commands. These commands are wrapped in the firmware functions `read_from_UART` and `send_to_UART`, whereas the read byte or the byte to send is passed in the register `UART_data`, respectively.

Command	Parameters	Description
`0x10`	1 address byte	Read 4 data bytes from the specified address from the timing generator control memory
`0x20`	1 address byte + 4 data bytes	Write 4 data bytes to the specified address into the timing generator control memory
`0x40`	None	Arm the timing generator, i.e., let it respond to trigger edges
`0x80`	None	Return the status byte, including the status of the external input pins in bit 2 ... 5
`0x90`	1 byte	Set the values of the external output pins. Only the lower half of the parameter byte is used.
`0xf0`	None	Reset the timing generator
`0x50`	1 byte	Set the prescaling factor for the clock fault generator
`0x60`	1 byte	Set the fine phase shift for the clock fault generator
`0x70`	1 byte	Set the coarse phase shift for the clock fault generator
`0xa0`	1 byte	Set the low voltage for the power fault generator
`0xb0`	1 byte	Set the high voltage for the power fault generator

Table 3.1.: UART control commands interpreted by the fault injection FPGA.

The main loop of the embedded code dispatches the commands issued by the PC

based on the first byte received when the controller waits for instructions in its idle state. Based on this byte, subroutines controlling the specific functions of the fault injection FPGA are called which are in turn responsible for handling further parameters. Tab. 3.1 summarises the available commands, however, note that at this point, the actual meaning of the parameters might remain unclear. They will be explained in the following sections where the functionality of the corresponding blocks is covered.

API On the PC side, the RS232 communication and the FPGA commands are wrapped in the class `fault_fpga`, which provides functions for all commands in Tab. 3.1, and additional methods that encapsulate low-level tasks, such as directly writing the parameters for specific pulse waveforms to the timing generator configuration memory. The class is utilised as follows:

```
fault_fpga fpga;

// Open connection to FPGA connected to COM1
fpga.open("/dev/com1");

// Initialise
fpga.init();

// Perform configuration
// ...

// Arm the FPGA
fpga.arm();

// Wait for device being armed
do {
    status = control.getStatus();
    ::usleep(1000);
} while(!(status & 1 << fault_fpga::FPGA_ARMED));


// Wait for device being ready again
do {
    ::usleep(1000);
    status = control.getStatus();
} while(!(status & 1 << fault_fpga::FPGA_READY));
```

```
// Close the connection
fpga.close();
```

Timing Controller

As we aim at constructing a flexible device, we separate the generation of the *fault trigger* `inject_fault`, i.e., the signal that starts a fault injection, from the logic responsible for controlling a specific fault module. This functionality is implemented in the VHDL block `timing_controller_waveform`, which is connected to a dual-port configuration memory `waveform_ram` realised as a BlockRAM. The `waveform_ram` holds 256 4 byte words for general configuration data and the sequence of pulses. The structure of the memory is depicted in Tab. 3.2. Note that currently the timing is static, as the specified pulse waveform is generated on a rising edge of an *external trigger* signal `trigger`, which can be, e.g., produced by the DUT and also serve to start an oscilloscope recording.

Address	Description
`0x00`	Configuration word holding the polarity and the number of configured pulses.
`0x01`	Reserved for future use.
`0x02`	Delay of the first pulse in FPGA clock cycles, with respect to the rising edge of the external trigger, where a value of zero corresponds to the minimum offset of four cycles.
`0x03`	Length of the first pulse in FPGA clock cycles, where a value of zero corresponds to the minimum lenght of one cycle.
`0x04`	Delay of the second pulse, with respect to the end of the first pulse, where a value of zero corresponds to the minimum offset of three cycles.
...	...

Table 3.2.: UART control commands interpreted by the fault injection FPGA.

The memory is accessable both from the microcontroller, allowing for configuring the fault injection timing via the PC interface, and from the timing controller, which generates the specified waveform and passes it to the actual fault modules. On the controlling PC, the timing is set by adding pulses to the FPGA control object `fault_fpga`. The corresponding methods are summarised by the following enumeration.

addPulse() Appends a pulse to the configuration memory, where the offset with respect to the previous pulse and the length of the pulse itself are both specified in ns.

clearPulses() Clears all pulses currently configured.

setPolarity() Sets the logic value of `inject_fault` in its active state, i.e., while a pulse for a fault injection is generated.

General Purpose Input/Output Pins

As mentioned in Sect. 3.5.1, the FPGA provides both four general purpose *Input/Output* (I/O) pins which can be utilised for, e.g., executing low-speed digital communications protocols, resetting the DUT (for instance, if a fault injection caused the device to crash) or reading the state of external inputs for detecting a successful fault injection. The methods provided by `fault_fpga` are:

getUserInput() Gets the state of the four input pins, encoded in the four low bits of the return value.

setUserOutput() Sets the state of the four output pins, where only the four low bits of the parameter are used.

Configuration File

To simplify the use of the fault injection FPGA together with `measurement_app`, a section in the measurement configuration file controls the respective parameters. Currently, only a `voltage_sweep` is supported, i.e., iterating over the offset, width and voltage level for faults generated by the power fault module described in Sect. 3.5.2. Aside the specification of the serial port for communication, the upper (`*_high`) and lower bounds (`*_low`) and the step width (`*_step`) for all mentioned parameters are set.

```
# COM port of FPGA
port = /dev/com1

# FPGA clock frequency in Hz (fixed in VHDL code, value cannot
# be manipulated here but MUST be CORRECT)
# default is currently 100 MHz
clock = 100000000

# voltage_sweep
mode = voltage_sweep

# high voltage (constant) in DAC counts (0 ... 255)
voltage_high = 175

# start voltage for low voltage sweep in DAC counts (0 ... 255)
voltage_low_start = 70
```

```
# end voltage for low voltage sweep in DAC counts (0 ... 255)
voltage_low_end = 80
# step size for low voltage sweep in DAC counts (min. 1)
voltage_low_step = 1

# start pulse offset in ns (min. 30 ns)
offset_start = 1330
# end pulse offset in ns (min. 30 ns)
offset_end = 2530
# step size for pulse offset in ns (min 10 ns)
offset_step = 100

# start pulse width in ns (min. 10 ns)
width_start = 2010
# end pulse width in ns (min. 10 ns)
width_end = 2210
# step size for pulse width in ns (min 10 ns)
width_step = 200
```

Interface between FPGA and Physical Fault Injection Module

The fault module, i.e., the *Printed Circuit Board* (PCB) containing the circuitry for realising physical faults, is attached to the FPGA board using the 40-pin expansion connectors A2 and B1. These sockets provide access to freely assignable FPGA pins, so that the interface to the fault module can be kept flexible. Therefore, we specify the connector pin allocation only partially in Tab. 3.3 and leave the rest open to satisfy the requirements of the concrete fault module. Note that in the pinout specification, `I` indicates an input to the FPGA, `O` an output from the FPGA and `P` a power supply or ground pin.

`CLK` is a separated clock pin designed to transport high frequency clock signals. It is surrounded by ground signals to minimise the disturbing influence on other signals and should be used for clocking external components where possible. Note that the frequency is not fixed, i.e., the pin can be used as desired.

Pin 40 functions as the trigger signal `trigger` for the internal timing generator and is shared between A2 and B1, that is, linked to the same FPGA pin. Pins 35 - 38 are defined as *Not Connected* (NC) and must be left open since they are bound to the FPGA programming interface. Pin 6 has an internal pulldown-resistor and should be connected to +3.3 V on the extension board to signalise the availability of a fault module. The user input and output pins 6, 8, ..., 20, 22 are dedicated to provide low-speed digital signals accessible from the controlling PC.

In the present version of the FPGA design, dynamically switching the functionality of the remaining pins based on the connected fault board type is not supported. The main reason is that this functionality would require (possibly wide) multiplexers at these pins – rendering the timing more difficult – and a way to recognise the module type, e.g. a small EEPROM or a hard-wired identifier.

Therefore, we use a static pinout, with the power fault module connected to A2 and the clock fault module to B1. The most straightforward way to support other configurations is to generate FPGA programming bitstreams for each desired variant and load the corresponding file via the programming interface before booting the device. Note that this could even be done dynamically on the controlling PC using the appropriate Xilinx command-line tools, so little flexibility is given up with respect to most part of the concrete applications.

		Pin	Pin		
GND	P	1	2	P	+5 V
+3.3 V	P	3	4		
		5	6	I	Device plugged
		7	8	I	User Input 0
		9	10	I	User Input 1
		11	12	I	User Input 2
		13	14	I	User Input 3
		15	16	O	User Output 0
		17	18	O	User Output 1
		19	20	O	User Output 2
		21	22	O	User Output 3
		23	24		
		25	26		
		27	28		
		29	30		
GND	P	31	32	P	GND
CLK	O	33	34	P	GND
NC		35	36		NC
NC		37	38		NC
NC		39	40	I	Trigger

Table 3.3.: FPGA to fault module connector pinout.

3.5.2. Power Fault Module

As described in Sect. 2.2.2, power faults can be both triggered by negative (i.e., reduction of the supply voltage) and positive glitches (i.e., increase of the supply voltage). For maximum flexibility in this respect and for fine control over the actual waveform, we have chosen a *Digital-Analogue Converter* (DAC) based approach, as depicted in Fig. 3.7.

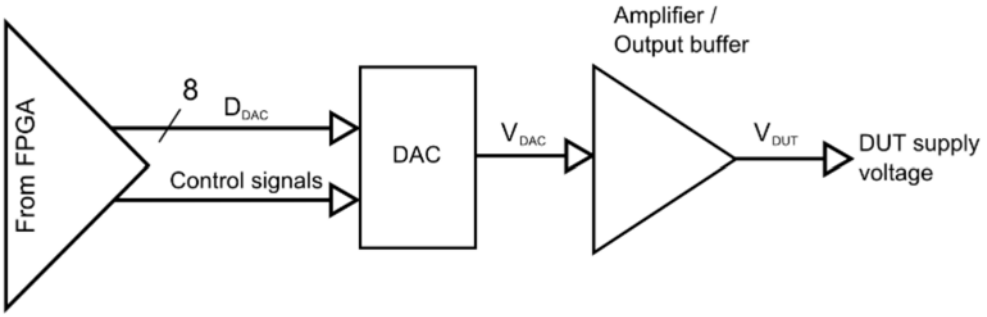


Figure 3.7.: Structure of power fault module.

The voltage V_{DAC} at the DAC output pin can be controlled via an 8-bit bus, passing a binary-encoded number $D_{DAC} \in \{0, \ldots, 255\}$. In the following, we occasionally refer to this value as *DAC counts*. V_{DAC} is then given as $V_{DAC} = \frac{D_{DAC}}{255} \cdot V_{DAC,max}$ where $V_{DAC,max}$ denotes the maximum output voltage.

Because the considered DAC ICs generate voltages of max. ≈ 1 V and cannot supply output currents > 20 mA, an additional output amplifier is required to provide higher voltages and greater driver strength. This part of the circuit has to be designed with the speed requirements for faults in mind. Here, the FPGA and the DAC run at 100 MHz, so the output amplifier has to match this specification and be able to pass through the waveform with as little distortion as possible.

API Since the timing is covered by the configuration of the timing controller, only the respective voltage levels need to be set by the controlling PC. Note that our current implementation is restricted to pulses with an upper and a lower voltage, however, could easily be extended to output arbitrary pulse forms due to the DAC approach. The API call for specifying the pulse voltages is `setDacVoltages()`, which takes the lower and the upper voltages encoded as 8-bit DAC counts.

Implementation Details

A PCB has been designed with the structure introduced above. The used DAC is the AD9708 manufactured by Analog Devices [Ana09b], capable of running at max. 125

MSamples/s. The output amplifier is a two-stage design, built with the current-feedback[6] *Operational Amplifier* (OP) AD8012 by Analog Devices [Ana09a] providing a theoretical bandwidth of 150 MHz at a gain of +2 and a slew rate[7] of 2250 V/μs. A simplified schematic of this output stage is given in Fig. 3.8. The complete schematic is given in Appendix C.1. In the following we discuss the circuit and its disavantages. Then, we present an improved revision that overcomes these problems.

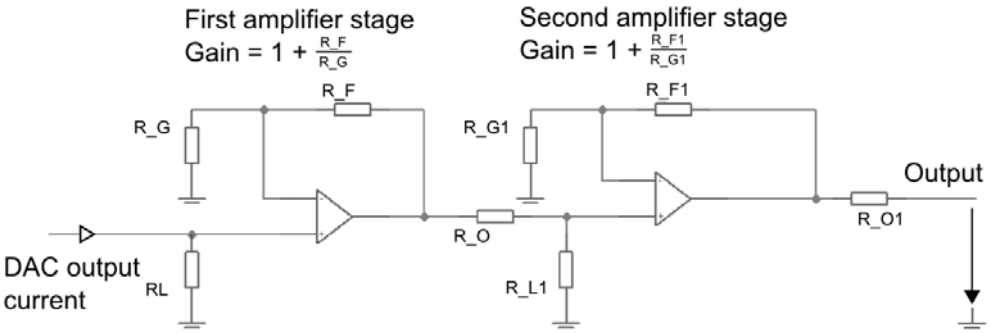


Figure 3.8.: Output stage of power fault module, revision 1.

Fig. 3.9 shows a negative voltage glitch ($\approx 4\,\mathrm{V} \rightarrow 0\,\mathrm{V}$) with the minimum width of 10 ns measured at the output of the DAC. This pulse is then amplified by the AD8012 in a non-inverting configuration by 4, resulting in the waveform given in Fig. 3.10. Note that both signals have been recorded using a probe set to x10 attenuation to minimise the influence of the probe capacitance and rise/fall times[8].

While the DAC is able to drive its output completely low, the OP cannot fully discharge the load capacitance within the required time. The primary reason for this behaviour is that the OP does not provide a *rail-to-rail* output, i.e., is unable to produce voltages close to its supply lines. Since the negative supply voltage is connected to ground, the minimum voltage is thus $\approx 1.3\,\mathrm{V}$. Although this is not problematic for the practical experiments described in Sect. 4.2 (because the required minimum voltage is greater), it somewhat limits the general applicability of the power fault module.

Besides, two other problems are present: First, the max. output amplitude is $\approx 4\,\mathrm{V}$, again partially restricting some applications, although most embedded devices can be operated at 3.3 V. Second, the clock signal of the DAC has disturbing influence on the supply voltage of the DAC, presumably because the signals are routed in proximity. This effect can be mitigated by only clocking the DAC when new values need to be written,

[6] For current-feedback amplifiers, the bandwidth is not in inverse proportion to the gain

[7] The slew rate indicates the maximum rate of change of the output voltage

[8] According to the manufacturer [TES], the used probe TESTEC TT-HF 312 has a rise time of 1.2 ns, an input resistance of 10 MΩ and a capacitance of 15 pF for x10 attenuation

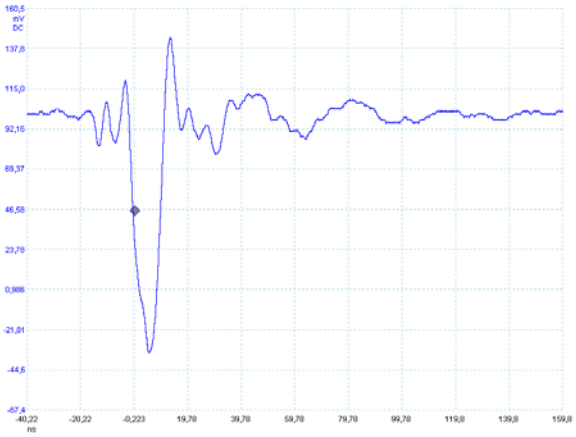


Figure 3.9.: Full-scale 10 ns negative voltage glitch at DAC output, x10 probe.

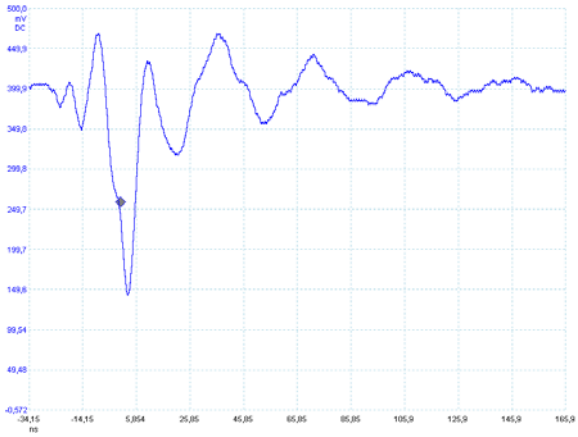


Figure 3.10.: Full-scale 10 ns negative voltage glitch at first amplifier output, x10 probe.

which is sufficient for the case of a simple pulse as output waveform (where most of the time the output is kept constant).

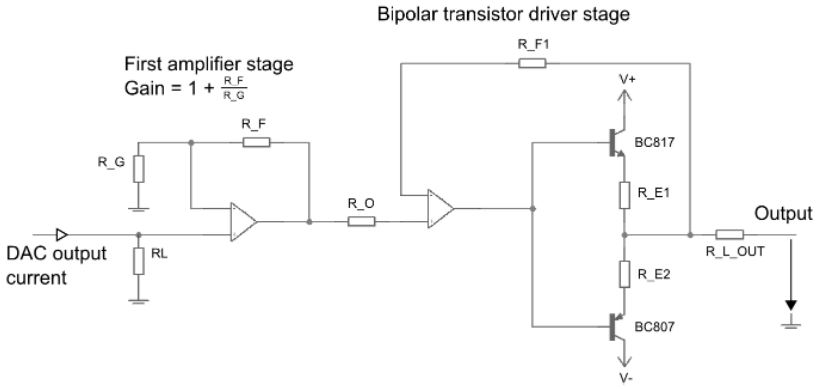


Figure 3.11.: Output stage of power fault module, revision 2.

To overcome these problems, a second revision has been designed. To increase the maximum amplitude up to ≈ 5.5 V, the output amplifier gain is set to ≈ 4.7. Additionally, a bipolar transistor-based output stage according to [Kug03], p. 258, has been implemented, which enables output currents of max. 150 mA at a supply voltage of 7.5 V. By selecting different transistors, this value could be further improved if required. The second reason for the redesign of the output stage is the inability of the first module version to fully discharge a load within the set time boundaries. Therefore, this time, the negative supply voltage of the OP is separated from ground, enabling bipolar operation

and in consequence output voltages down to 0 V. The second revision of the output stage is depicted in Fig. 3.11. The complete schematic can be found in Appendix C.2.

The noise is lowered (but not completely eliminated) by improved layout, capacitive decoupling of the DAC supply voltage pins and the reduction of the FPGA driver strength for the clock line to 10 mA, which turns out to be the smallest working setting. Fig. 3.12 and Fig. 3.13 depict a 10 ns full-scale pulse generated with the revised power fault module, recorded at different points of the circuit.

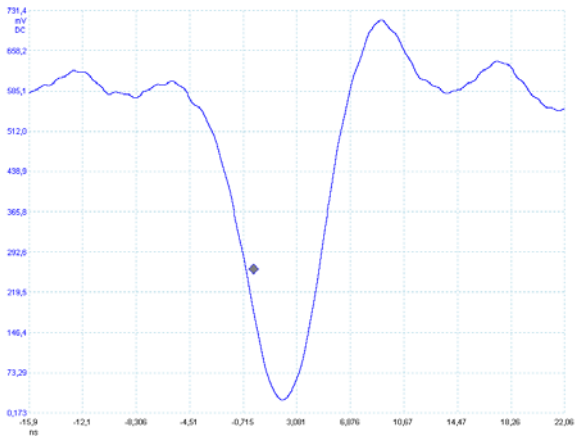


Figure 3.12.: Full-scale 10 ns voltage glitch at amplifier output, module revision 2, x10 probe.

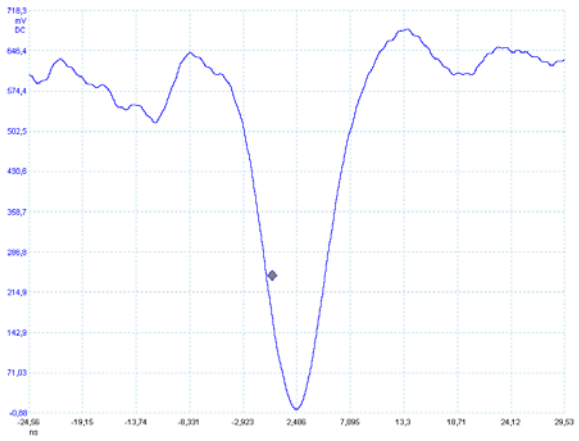


Figure 3.13.: Full-scale 10 ns voltage glitch after transistor output stage, module revision 2, x10 probe.

3.5.3. Clock Fault Module

Clock faults as introduced in Sect. 2.2.2 are generated by a small, temporary variation of the period of the clock signal. For maximum flexibility, a module for this type of fault has to provide:

- A wide range of output frequencies, especially covering the range of embedded systems and
- precise control over the duration the clock signal is set.

Our approach makes use of the DCM of the Xilinx FPGA which is able to generate a clock signal with very fine control over its phase shift. By outputting both an unshifted and a shifted clock and combining these signals logically with external circuitry, several useful waveforms can be created.

As in the present implementation, available gates would have been unused, we decided to include three other waveforms. Aside from stretching a clock cycle, moreover short

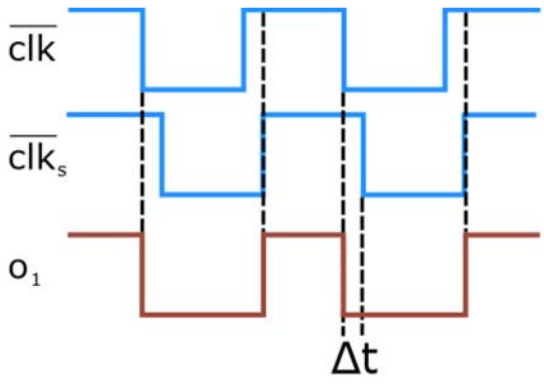


Figure 3.14.: Clock fault signal $o_1 = \overline{clk} \wedge \overline{clk_s}$ for shortening clock cycles.

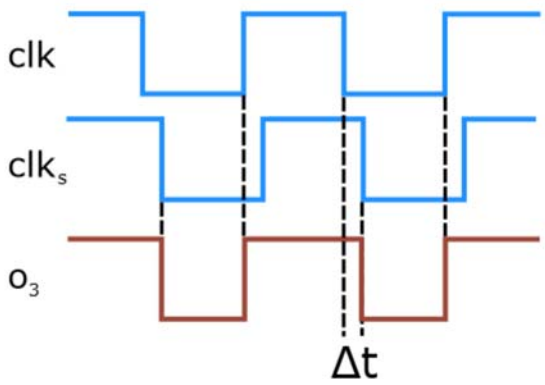


Figure 3.15.: Clock fault signal $o_3 = clk \vee clk_s$ for stretching clock cycles.

(positive and negative) pulses with a width smaller than the min. 10 ns provided by the power fault module (cf. Sect. 3.5.2) are available. This can be advantageous for DUTs running at high frequencies. Nevertheless, the control over the waveform is less sophisticated compared to the power fault module (e.g., with regard to voltage levels and fast changes of the pulse width).

Let clk denote a clock signal, clk_s this signal shifted by Δt, $\overline{\cdot}$ logical inversion, $\wedge$ logical AND and $\vee$ logical OR. The outputs of the clock fault module are constructed as given in Eqn. 3.1. All signals are illustrated in Fig. 3.14, Fig. 3.16, Fig. 3.15 and Fig. 3.17.

$$\begin{aligned} o_1 &= \overline{clk} \wedge \overline{clk_s} \\ o_2 &= clk \wedge \overline{clk_s} \\ o_3 &= clk \vee clk_s \\ o_4 &= \overline{clk} \vee clk_s \end{aligned} \tag{3.1}$$

Implementation Details

The generation of the shifted clock signals clk_s is performed by the FPGA using a combination of a fine phase shift, followed by clock (down-)scaling and a coarse phase shift. The other needed signals (clk, $\overline{clk}$, $\overline{clk_s}$) can be obtained by directly outputting the (downscaled) clock and inverted versions of the appropriate signals, respectively.

As mentioned above, the fine shift is realised with the phase shift function of the Xilinx DCM, which allows a clock to be shifted by $\frac{1}{256}$th of the clock period. The input

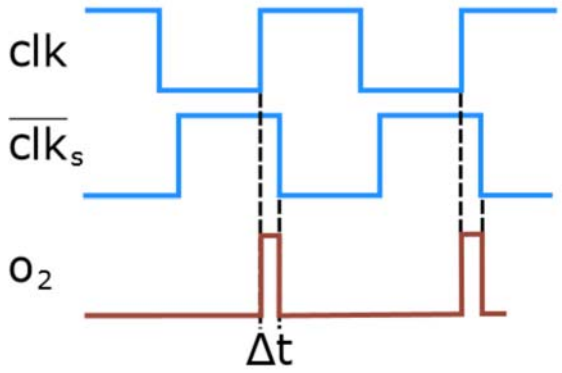


Figure 3.16.: Clock fault signal $o_2 = clk \wedge \overline{clk_s}$ for short positive pulses.

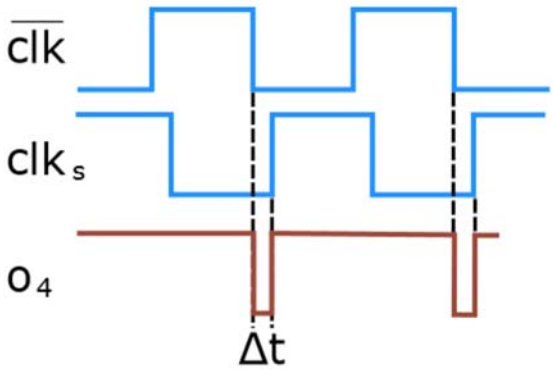


Figure 3.17.: Clock fault signal $o_4 = \overline{clk} \vee clk_s$ for short negative pulses.

to the DCM is the global system clock clk_{sys}, the output $clk_{sys,\,s}$ (shifted by Δ_{fine}), both running at 100 MHz.

clk_{sys} and $clk_{sys,\,s}$ are passed to a prescaler (and coarse phase shifter) implemented in the VHDL component `clock_divider`, which toggles the output clock when an internal counter reaches half of the configured prescaling factor. The coarse phase shift by Δ_{coarse} is accomplished by pre-loading the internal counter on startup. This way, clk_s can be shifted in multiples of the system clock frequency, i.e., in steps of 10 ns, with respect to clk.

Currently, the 100 MHz system clock frequency can only be divided by even factors (without affecting the 50 % duty cycle[9]), allowing to realise the prescaler with essentially one counter and one comparator. In order to enable the division by odd factors, a design reacting to both the falling and rising clock edge is necessary, as described in [Boo]. The downscaled clocks clk and clk_s (and their inverted counterparts) are then routed to FPGA output pins connected to the actual fault module PCB. Fig. 3.18 summarises the complete process.

For inducing a clock fault, the clock shall typically be modified once or a few times and then return to its "normal" mode, i.e., 50 % duty cycle. Hence, the modification of the phase shift (which controls the variation for o_1 and o_3 and the pulse width for o_2 and o_4, respectively) must be precisely timed within one or few (half-)clock cycles.

We apply the VHDL timing module described in Sect. 3.5.1 to generate a pulse waveform according to the desired fault timing. This signal is passed to the input `clock_shift_en` of `clock_generator`, controlling whether a normal (`clock_shift_en = 0`) or a faulted (`clock_shift_en = 1`) clock signal is generated. Note that the actual clock for the DUT is present *after* the logical combination of clk and clk_s. Hence, if these

[9]The duty cycle is the percentage of one period during which the signal is high

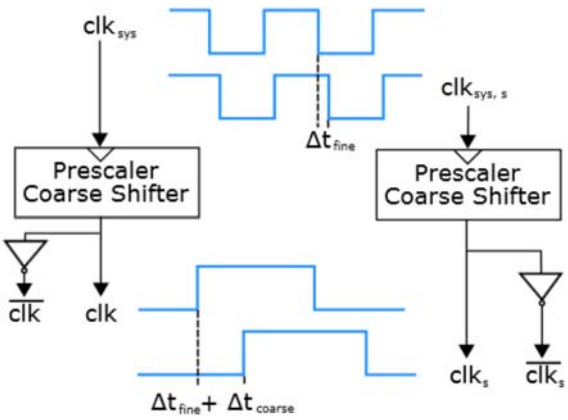


Figure 3.18.: Clock signal shifting and prescaling on FPGA.

signals are in-phase, i.e., $\Delta t = 0$, a normal clock is generated. The modification only is effective if $\Delta t > 0$, thus, changing the phase shift Δt of the FPGA outputs controls whether the clock signal is faulted or not.

Directly dynamically modifying the phase shift generated by the DCM is not an option, as it is a rather time-intensive task. The phase adjustment has to be performed in steps of $\frac{1}{256}$th of the clock period and moreover, each step can last up to 103 cycles before becoming effective [Xil08b]. Therefore, we propose a different method, by multiplexing - based on the `clock_generator` input `clock_shift_en` - the shifted clock outputs as follows:

```
if clock_shift_en = '1' then
    clock_out_shifted <= clk_div_shift;
    clock_out_shifted_not <= not clk_div_shift;
else
    clock_out_shifted <= '0';
    clock_out_shifted_not <= '1';
end if;
```

This assignment ensures that if `clock_shift_en = 0`, o_1 and o_3 are unmodified, prescaled clocks, while o_2 and o_4 are constantly 0 and 1, respectively. This is equivalent to setting $\Delta t = 0$, however, substantially reduces the FPGA logic overhead.

On the external board, the logical operations proposed above are performed by discrete, high speed CMOS ICs. According to the datasheets [Pot09], these ICs can be operated at frequencies above 1 GHz, enabling precise adjustment of the signal timing. An example output signal for o_1 (i.e., a slightly shortened clock cycle) is depicted in Fig. 3.19, with the prescaler set so that the output clock frequency is 16.67 MHz.

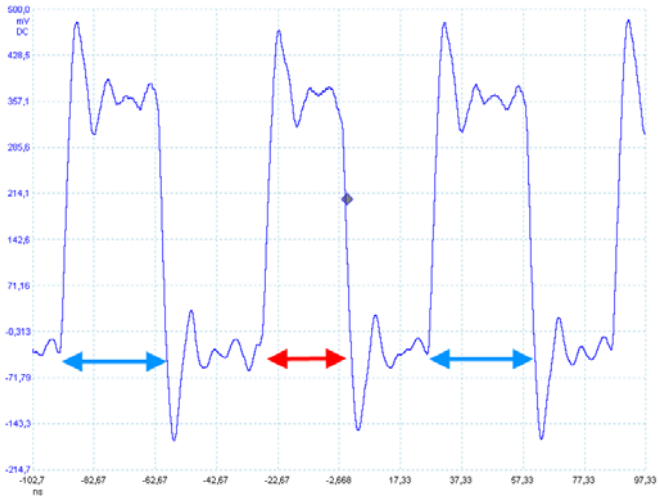


Figure 3.19.: Clock fault on 16.67 MHz clock signal, x10 probe, 20 ns/time division.

For the case that the clock module is used to drive a larger load (the logic ICs can only source 5 μA), an output stage is included on the PCB, built basing on the Analog Devices AD8045 voltage feedback OP [Ana04], capable of supplying up to 55 mA. Currently, the OP is configured as a buffer (i.e., with amplification of +1), however, additional pads have been reserved in order to enable operation as a non-inverting amplifier. Refer to Appendix C.3 for the complete schematic.

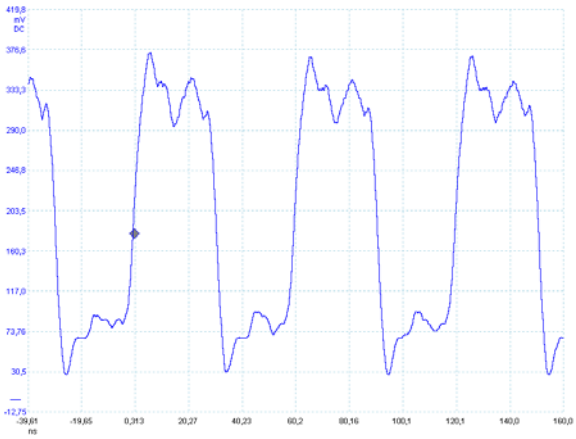


Figure 3.20.: 16.67 MHz clock signal after output buffer, x10 probe.

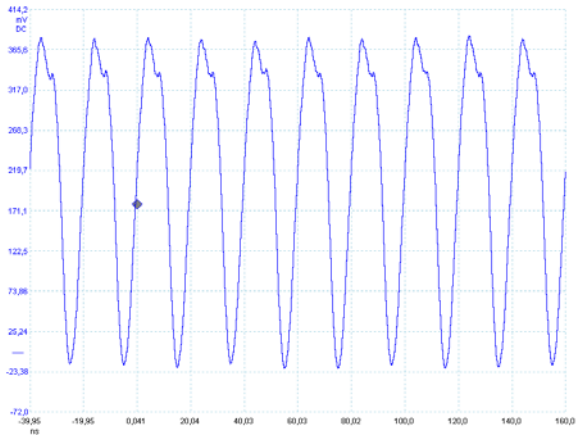


Figure 3.21.: 50 MHz clock signal after output buffer, x10 probe.

This enables the generation of voltages greater than 3.3 V, which is useful if the clock fault module is employed to produce short pulses, i.e., when o_2 and o_4 are used. Example output signals of the buffer stage for clocks of 16.67 MHz and 50 MHz are given in Fig. 3.20 and Fig. 3.21, demonstrating the ability to generate waveforms in the

specified frequency range.

API The configuration on the PC-side is performed by three methods that control the prescaling factor and phase shift, as outlined in the following enumeration. The timing of the fault is specified as for the voltage fault module, i.e., a pulse determining the offset and the duration of the clock modification is configured using `addPulse()`.

setPhase() Sets the fine phase shift in $\frac{1}{256}$ ns steps, encoded as 8-bit integer, with a maximum shift of $\frac{127}{256}$ ns.

setPhaseCoarse() Sets the coarse phase shift in FPGA clock cycles, i.e., 10 ns steps

setPrescaler() Specifies the division factor of the output clock, where only even factors yield a 50 % duty cycle signal, as mentioned in Sect. 3.5.3.

4. Practical Attacks

In this chapter, we demonstrate the capabilities of the developed framework by conducting some of the attacks described in Chap. 2 on real devices. We begin with two key recovery CPA attacks on different DUTs and continue with the injection of power faults on a popular microcontroller.

4.1. Side-Channel Analysis

As a first step towards the analysis of real-world devices, a CPA on a software implementation of the AES on a microcontroller-based smartcard is performed. Next, we target a real-world contactless smartcard featuring a 3DES hardware engine.

4.1.1. Microcontroller-based Smartcard

The microcontroller, an 8-bit Atmel Atmega 163 [Atm03], is embedded in a plastic card. Its interface resembles that of secure smartcards, i.e., is ISO 7816-compliant, yet, the controller is well-known to leak processed data. Its power consumption is highly correlated to the Hamming weight of the processed data, making the device a reasonable proof-of-concept and verification target for our framework, before proceeding with more sophisticated attacks.

We target a known, unprotected, straightforward AES implementation and record 1000 power traces at a sample rate of 100 MSamples/s during the encyption of random, uniformly distributed plaintexts with AES. To communicate with the DUT, we use the smartcard reader introduced in Sect. 3.3.1 to send the plaintext. The microcontroller then performs the AES encryption and returns the ciphertext to the controlling PC.

The code on the microcontroller provides a trigger signal to start the measurement by generating a short pulse on the data pin of the smartcard interface just before executing the encryption[1]. The power consumption is acquired by inserting a small resistor into the ground connection of the DUT using the smartcard adaptor introduced in Sec. 3.4.2.

[1] The pulse if sufficiently short so that it is not recognised as valid data by the reader but can be detected by the Picoscope pulse width trigger

The resistor value was adjusted so that the amplitude of the side-channel signal matches the minimal input range of ± 100 mV of the oscilloscope.

We target the output of the AES S-Box in the first round and thus define a CPA prediction function as the Hamming weight of the 8-bit S-Box result. AES is designed to operate *byte-wise*, i.e., the first plaintext byte is xor'ed with the first subkey byte and subsequently passed to the S-Box[2]. For further illustration, cf. Fig. 2.1 and Alg. 5.

Fig. 4.1 shows the correlation coefficient as defined in Eqn. 2.2 for all 256 candidates for the first byte of the subkey K_0. The horizontal lines at ± 0.23 indicate the theoretical noise level $\frac{4}{\sqrt{L}}$ (cf. [MOP07]), where L is the number of traces. The correct (highlighted) subkey is already clearly distinguishable after processing 300 traces. In subsequent experiments, it turns out that 95 traces are sufficient so that the correct subkeys yields the largest correlation in the region of interest (zoomed in Fig. 4.2) and hence can be identified. Using the presented framework, a full-key recovery, i.e., the recording and evaluation of the traces can be performed within a few minutes.

Because the results are conclusive for the first byte (and because all other bytes are processed analogical), we omit further evaluations to recover the other subkey bytes. As mentioned above, this *white-box attack*[3] on the AES implementation primarily serves as a preparative step for more challenging applications of the measurement environment. However, by successfully reproducing this straightforward side-channel attack, we demonstrate and verify the functionality provided by the framework with respect to side-channel analysis.

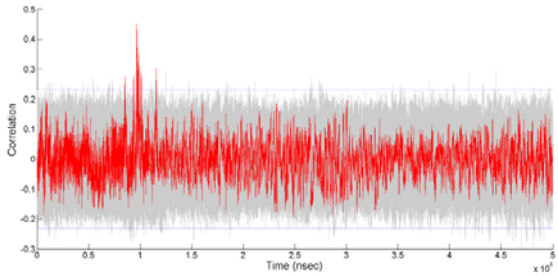


Figure 4.1.: Correlation after $L = 300$ traces for first byte in first round of AES, Hamming weight.

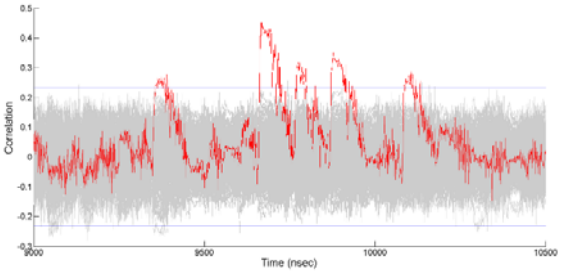


Figure 4.2.: Correlation after $L = 300$ traces for first byte in first round of AES, Hamming weight (zoomed).

[2] AES uses the same S-Box throughout the whole encryption process

[3] We know the code of the software implementation and the appropriate leakage model for the microcontroller

4.1.2. Commercial Contactless Smartcard

In this section, we turn towards a more complex scenario and analyse a commercially available contactless smartcard, extending our previous results [Osw08, KOP09]. This time, we are facing a *black-box situation*, i.e., do not know anything about the implementation of the cipher, existent countermeasures etc. Therefore, extensive profiling is necessary in preparation for a key recovery attack.

The DUT is an ISO 14443-compliant RFID device [iso01a, iso01b], operating at 13.56 MHz. The PICC features a challenge-response authentication protocol which relies on a symmetric block cipher, involving a 112 bit key k_C that is shared between the PICC and the PCD. For the cipher, a 3DES using the two 56 bit halves of $k_C = k_1||k_2$ in Encrypt-Decrypt-Encrypt mode according to [FIPb] is implemented. After a successful authentication, the subsequent communication is encrypted with a session key.

Since the smartcard IC is embedded in a plastic card and since – in contrast to the previous section – the power is drawn from the field of the reader rather than supplied via an electrical contact, direct measurements of the power consumption are impossible without tampering with the DUT. Thus, in order to allow a non-invasive analysis, we measure the EM field in proximity to the IC and perform a DEMA of the 3DES implementation. This process can be split up into the following steps, which we will detail in this section:

1. Find a suitable trigger point.
2. Profile the device and locate the 3DES encryption.
3. Focus on one 3DES encryption.
4. Perform the EM analysis of the 3DES encryption.

Challenge-Response Authentication Protocol

Using the RFID reader mentioned in Sect. 3.3.2, we implement the whole authentication protocol, but however, focus on the step relevant for our analyses as depicted in Fig. 4.3, where $\mathrm{3DES}_{k_C}(\cdot) = \mathrm{DES}_{k_1}\left(\mathrm{DES}^{-1}_{k_2}\left(\mathrm{DES}_{k_1}(\cdot)\right)\right)$ denotes a 3DES encryption involving the key $k_C = k_1||k_2$. The values B_1 and B_2 have a length of 64 bit and are encrypted by the PICC during the mutual authentication. B_2 originates from a random number previously generated by the PICC and is always encrypted by the PICC in order to check the authenticity of the PCD[4]. B_1, a random value chosen by the PCD that serves for authenticating the PICC to the PCD, is mentioned here for completeness only and is not required in the context of our analyses.

[4]The protocol will abort after the encryption of B_2, in case its verification is not successful.

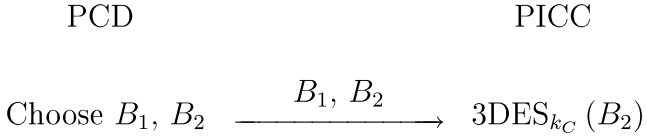


Figure 4.3.: Exerpt of the authentication protocol relevant for an attack.

By performing a full authentication and reproducing the responses[5] of the contactless smartcard under attack on the PC, we verify that a standard (3)DES [FIPb] is used for the encryption of the challenge according to Fig. 4.3. We further observe that the card unconditionally encrypts any value B_2 sent to it, hence we can freely choose the plaintext.

For the CPA described in the following, we send random, uniformly distributed plaintexts for B_2 and attack the first DES round[6]. Note that it is not necessary to completely power-down and reset the PICC after a failed authentication attempt. Instead, it is possible to send as many subsequent authentication enquiries as desired, thus avoiding the need to perform the complete, time-consuming initialisation sequence specified in [iso01a] for each trace. The trigger signal is provided by our custom reader, generating a falling edge on an I/O pin after the last bit of the challenge has been transmitted to the PCD.

Requirements of a Full-Key Recovery for 3DES

The DUT supports both a DES and 3DES for encryption, hence – depending on the selected mode – there exist certain implications with regard to the complexity of a full-key recovery. In the case of a normal DES, revealing the subkey of the first round by means of CPA yields 48 of the 56 key bits, so the remaining 8 bit can easily be found by performing 2^8 trial authentications.

If a dual-key 3DES is employed, a successful CPA on the first round of the first DES recovers 48 bit of the total 112-bit key. To recover the second key k_2, a CPA can be mounted on the second DES iteration, i.e., the decryption with k_2. As the input to this operation has to be known, a complete recovery of the first key k_1 is necessary. However, as mentioned, a CPA on the first round leaves 8 bit unknown.

Thus, there exist two alternatives for a full-key recovery: Either, 2^8 separate CPAs are performed for the the 2^8 possible candidates for k_1[7], or the remaining 8 bit are retrieved

[5] Note that in this context the secret key of the implementation can be changed by us and is hence known.

[6] Attacking the final round is impossible in this context as the PICC never outputs the result of the encryption of B_2

[7] Only one of these attacks has the correctly encrypted input data and hence shows significant correlation peaks

by performing an attack on the subsequent rounds of the first DES. The remaining 8 bit of k_2 can then be exhaustively searched in both cases, similar to the case of a single DES.

Trace Pre-Processing

We record traces between the last bit of the command sent by the reader and the first bit of the answer of the card, both with and without the analogue pre-processing filter introduced in Sec. 3.4.2. However, in both cases the signals do not expose any distinctive pattern, hence, additional digital pre-processing is applied in order to identify interesting patterns useful for a precise alignment of the traces. On the basis of the RFID power model introduced in [Osw08], we assume that the power consumption of the smartcard modulates the amplitude of the carrier signal at frequencies much lower than the 13.56 MHz carrier frequency, which is justified by a preliminary spectral analysis and the well-known fact that the on-chip components (such as capacitances, resistors, inductances) typically imply a strong low-pass filter characteristic.

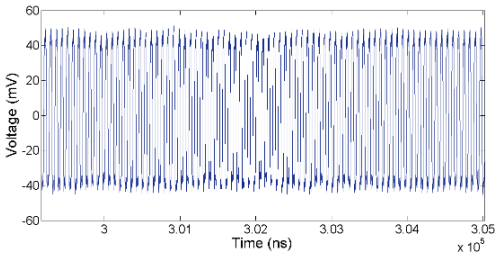


Figure 4.4.: Raw trace of 3DES encryption with analogue filter (zoomed).

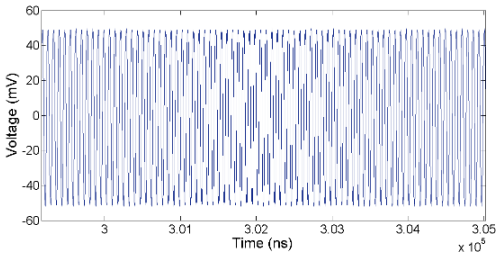


Figure 4.5.: Raw trace of 3DES encryption without analogue filter.

Digital Amplitude Demodulation In order obtain the relevant side-channel information, we record raw (undemodulated) traces and perform the demodulation digitally, using the straightforward incoherent demodulation approach depicted in Fig. 4.6, cf. [Sha06]. The raw trace is first rectified, then low-passed filtered using an appropriate digital filter. Next, an additional high-pass filter removes the constant amplitude offset resulting from the demodulation principle and low-frequency noise. Suitable values for the filter cutoff frequencies $f_{lowpass}$ and $f_{highpass}$ are determined experimentally and given in Sect. 4.1.2.

Figure 4.4 displays a demodulated trace ($f_{lowpass} = 2\,\text{MHz}$, $f_{highpass} = 50\,\text{kHz}$) in which distinct patterns are visible, especially two shapes at 240 μs and 340 μs, preceded and followed by a number of equally spaced peaks. For comparision, Fig. 4.7 shows a zoomed part of the same trace without demodulation. Fig. 4.8 and Fig. 4.5 originate

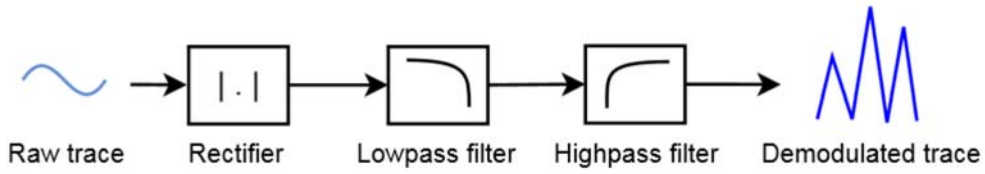


Figure 4.6.: Block diagram of incoherent digital amplitude demodulator.

from a trace recorded without the analogue filter described and demonstrate that our filter circuit effectively increases the amplitude of the signal of interest and reduces the noise level of the demodulated signal.

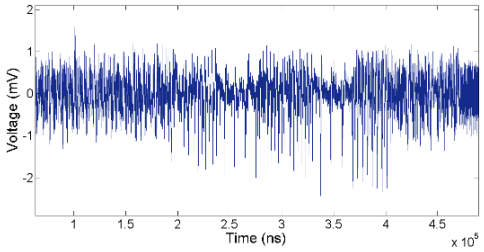


Figure 4.7.: Demodulated trace (50 kHz - 2 MHz) of 3DES encryption with analogue filter.

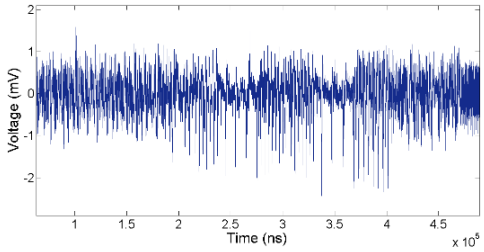


Figure 4.8.: Demodulated trace (50 kHz - 2 MHz) of 3DES encryption without analogue filter.

Trace Alignment

For precise alignment during the digital processing, we select a short reference pattern in a demodulated *reference trace*. This pattern is then located in all subsequent traces by finding the shift in time that minimises the squared difference between the reference and the trace to align, i.e., we apply a least-squares approach.

For devices with a synchronous clock, the alignment with respect to one distinct pattern is usually sufficient to align the whole trace. However, in our measurements we found that the analysed smartcard performs the operations in an asynchronous manner, i.e., the alignment may be wrong in portions not belonging to the reference pattern. The alignment has thus to be performed with respect to the part of the trace we aim to examine by means of CPA.

Having aligned the traces in time, further profiling of the DUT is mandatory before attempting to recover the key. Therefore, we first identify the part of the power trace during which the cryptographic operation takes place. Afterwards, the analysis of the

actual encipherment process can be accomplished, revealing the suitable power model and evaluating the level of protection provided by the DUT against side-channel attacks.

Locating the Data Bus Transfer of Plain- and Ciphertext

As the plaintext for the targeted 3DES operation is known and the ciphertext can be computed in a known-key scenario, we are able to isolate the location of the 3DES encryption by correlating on these values. From the profiling phase with a known key, we assume that the smartcard uses an 8-bit data bus to transfer plain- and ciphertexts: the corresponding values can be clearly identified from 2,000 - 5,000 traces using a Hamming weight model, as depicted in Fig. 4.9 and Fig. 4.10.

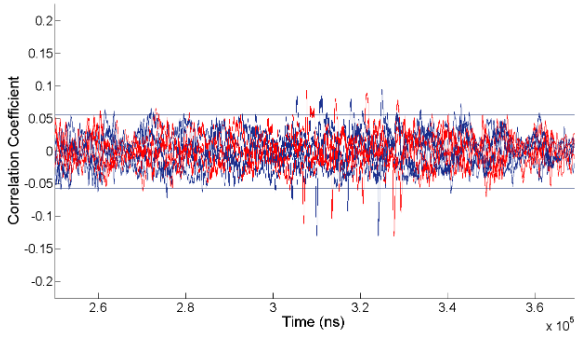


Figure 4.9.: Correlation coefficients for plaintext bytes (before targeted 3DES encryption) after 5,000 traces, Hamming Weight.

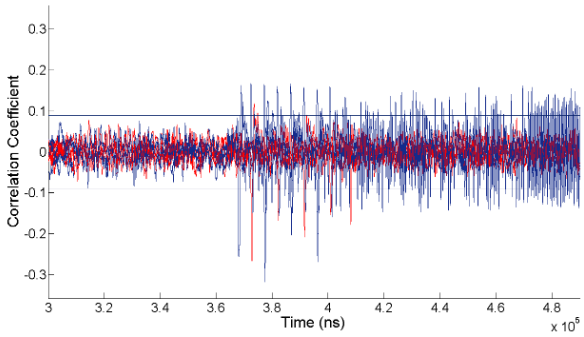


Figure 4.10.: Correlation coefficients for ciphertext bytes (after targeted 3DES encryption) after 2,000 traces, Hamming Weight.

This first result suggests that the smartcard logic is implemented on a microcontroller which communicates with a separate 3DES hardware engine over a data bus using precharged wires. This assumption is further supported by the fact that correlation with the plaintext bytes can be observed twice, but with reversed byte order. The microcontroller presumably first receives the plaintext bytes via the RFID RF frontend, byte-reverses it and transmits it over the internal bus to the encryption engine. The ciphertext is then sent back using the same byte order as for the second appearance of the plaintext.

From these (and additional) profiling observations, Fig. 4.11 was compiled, with the shape of the 3DES operation marked. The first 3DES encryption (3DES 1) results from a prior protocol step, the correlation with the correct ciphertext appears after the second 3DES shape only (labeled 3DES 2). We perform a CPA on the second 3DES, however, note that one may also decide to attack the first 3DES.

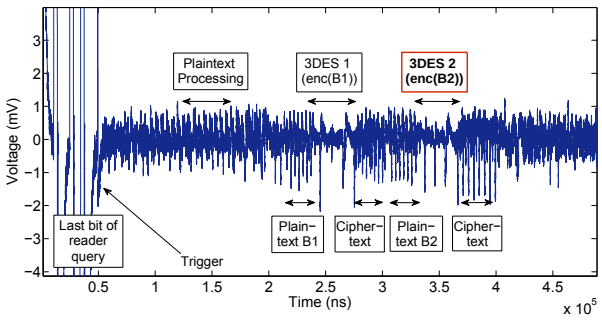


Figure 4.11.: Overview over operations in amplitude-demodulated trace.

DEMA of the 3DES Implementation

Having localised the interval of the 3DES operation from the position of the corresponding plain- and ciphertexts, we now focus on this part of the trace. Figure 4.12 shows a zoomed view of the targeted 3DES operation, filtered with $f_{lowpass} = 8\,\text{MHz}$ and $f_{highpass} = 50\,\text{kHz}$. The short duration of the encryption suggests that the 3DES is implemented in a special, separate hardware module, hence in the following we assume a Hamming distance model[8].

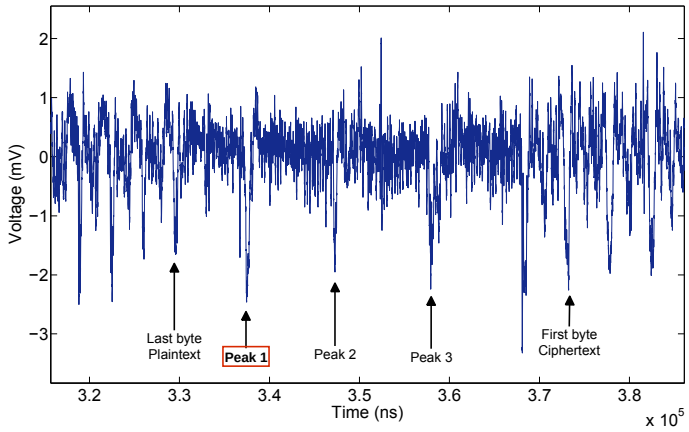


Figure 4.12.: Part of trace with 3DES encryption, filtered with $f_{lowpass} = 8\,\text{MHz}$, $f_{highpass} = 50\,\text{kHz}$.

Attack on First DES Iteration The three marked peaks in Fig. 4.12 seemingly appear at the begin of one complete single DES and are thus promising candidates as alignment patterns. Consequently, we conduct a CPA on demodulated traces aligned to each

[8]We also considered a Hamming weight model, however, did not reach conclusive results with it

of these peaks, where we consider the Hamming distance between the DES registers (L_0, R_0) and (L_1, R_1), i.e, the state before and after the first round of the first single DES. More specifically, for each S-Box, we predict the Hamming distance between the 4 S-Box output bits at the positions after the DES permutation layer P (i.e., the *output* of the first round) and the bits at the respective positions at the *beginning* of the first round.

For the alignment to the first peak, correlation peaks with maximum amplitude for the correct subkey candidate for all S-Boxes occur after 1,000,000 traces, as depicted in Fig. 4.13. The vertical lines mark the theoretical noise level $\frac{4}{\sqrt{L}}$, with L the number of traces. Note that he suitable alignment pattern and the prediction function for the CPA are experimentally determined in a semi-automatic manner. For this purpose, the ability of our framework to quickly test many possible CPA settings using appropriate configuration files turns out to heavily reduce the time required to find the correct parameters.

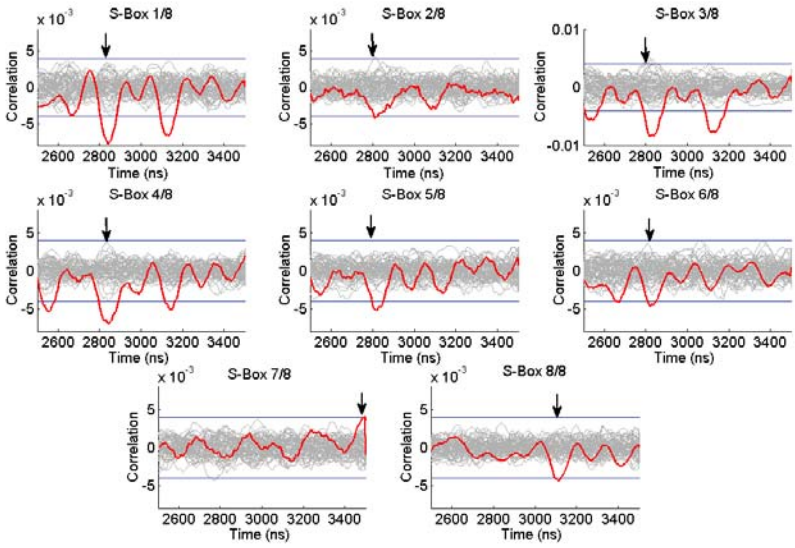


Figure 4.13.: Correlation coefficients for first DES, first round, after 1,000,000 traces, $f_{lowpass} = 8\,\text{MHz}$, $f_{highpass} = 50\,\text{kHz}$.

We observe that 1. the correlation for the output of some S-Boxes is significantly stronger than for others (e.g., for S-Box 1 and 3, for which the correct subkey can already be identified after 150,000 traces), 2. several peaks appear at different points in time for one S-Box and 3. the point of maximum correlation varies depending on the S-Box.

As the attack works (albeit after a large number of traces), we conclude that no masking scheme [MOP07] is used to protect the hardware engine. Rather than, we conjecture that hiding in time dimension is used, e.g., dummy cycles with no computation taking

place or similar measures might be inserted to prevent correct alignment of the traces. This assumption is justified by the above observation that more than one peak occurs in the correlation curve and further strengthened by the fact that when repeatedly sending the same plaintext B_2 to the smartcard, the shape of the DES operation and the position of the peaks depicted in Fig. 4.12 vary[9]. This behaviour can also be observed when completely resetting the DUT by switching off the field of the reader between two identical authentication attempts with the same plaintext B_2 and a fixed timing, i.e., with precisely defined points in time at which each command is sent. Hence, the randomisation of the computation does not seem to depend on parameters directly controllable by an adversary, e.g., the timing of the authentication protocol.

With our measurement setup, recording one million traces takes approx. two days, i.e., we achieve a rate of approx. 700 measurements per minute. The subsequent evaluation on a single high-end PC takes approx. one day. Having extensively profiled the DUT, we are able to focus on the relevant region of the EM trace and thus achieve substantial savings both with regard to disk space and processing time. Therefore, our attack is still feasible in a practical scenario, despite the considerable amount of traces.

Attack on Second DES Iteration Having recovered 48 bit of the first half k_1 of the key, we consequently attempt to attack the second iteration, i.e., the decryption part. We do not explicitly address the problem of recovering the complete key for the first DES iteration. We note that both approaches of Sect. 4.1.2 are feasible, even the exhaustive search using 256 CPAs. Hence, we assume the first key half to be known and use it in our experiments. Following the steps of the attack on the first iteration, the traces are this time aligned to the second peak. The result displayed in Fig. 4.14 is similar to that of the previous section and allows to determine the correct subkey candidate for all S-Boxes. Note that this time a smaller part of the traces is considered to reduce the computation time.

From these results, we conclude that a full-key recovery on the given real-world contactless smartcard is feasible. Having found the 3DES key, an adversary is for instance able to eavesdrop and decrypt the communication between a legitimate reader and the smartcard, read out and/or modify stored data, or create an cryptographically indistinguishable copy of the card. Depending on the key derivation scheme used in a concrete application scenario, the recovery of the key of a single device can have a severe impact on the security of the whole system, for instance, if the same symmetric key is used for a certain group or for all cards. In that case, an adversary gains complete access to all devices sharing the same key. Hence – considering the non-invasive full-key recovery attack presented in this section – it is of special importance to ensure that each contactless smartcard in a system shares a unique key with the reader by applying a secure key-derivation mechanism.

[9] This misalignment also hinders improving the SNR by means of averaging.

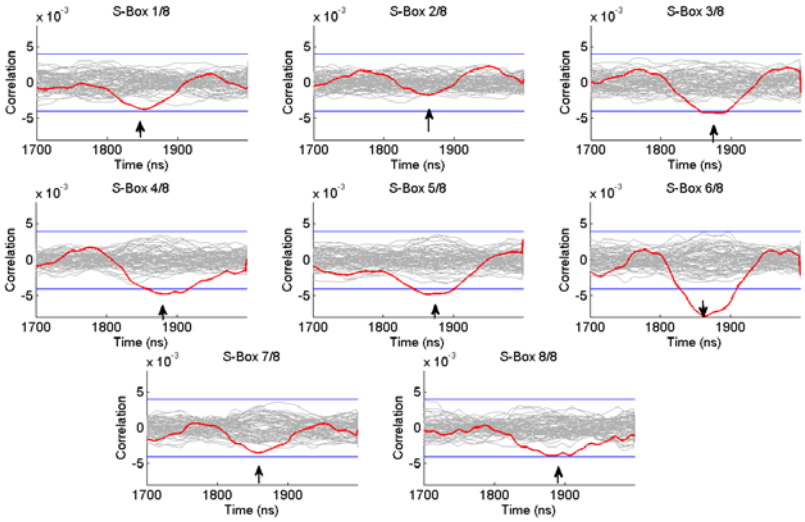


Figure 4.14.: Correlation coefficients for second DES, first round, after 1,000,000 traces, $f_{lowpass} = 8\,\text{MHz}$, $f_{highpass} = 50\,\text{kHz}$.

4.2. Fault Injection Attacks

After demonstrating the capabilities of the developed framework with regard to passive side-channel analysis, we address active fault injection and accordingly carry out an attack against a common 8-bit microcontroller, the PIC16F687 [Mic08]. For the following experiments, we use a simple PCB containing the microcontroller and pin headers connected to the supply voltage pin V_{cc}, the ground pin GND, the reset pin and a feq user-programmable inputs/outputs, e.g., to indicate the current status of the DUT. The supply voltage V_{cc} is set to 4 V.

The microcontroller is clocked by its internal oscillator, at a frequency of $f_{PIC} = 8\,\text{MHz}$. According to the datasheet [Mic08], one instruction cycle of length T_i takes four oscillator cycles, i.e., $T_i = 4 \cdot T_{PIC} = 500\,\text{ns}$, where $T_{PIC} = \frac{1}{f_{PIC}} = 125\,\text{ns}$. Most instructions are executed within a single instruction cycle, however, conditional instructions and jumps such as `goto` require two cycles and hence take $T_{goto} = 2 \cdot T_i = 1\,\mu\text{s}$.

4.2.1. Single Faults

In this section, the main focus is on the demonstration of the basic feasibility of fault injection techniques, not on the implementation of attacks against actual cryptographical algorithms. Consequently, we start with defining a simple test scenario, attempting to skip one instruction executed by the microcontroller. The microcontroller executes a simple program to detect that a fault has been successfully injected. After the initialization, a rising edge is generated on `PIN_TRIGGER`, triggering the fault injection FPGA.

Then, the status pin `PIN_STATUS` (connected to an FPGA input) is constantly set to high in a first infinite loop. In a subsequent infinite loop, the same pin is pulled low and additionally, another status pin `PIN_STATUS_2` is constantly toggled, thereby indicating whether the microcontroller is still alive.

```
main
    ; prepare: pin directions
    banksel OUTTRIS
    movlw  b'00000000'
    ; all pins are outputs
    movwf OUTTRIS

    ; set all outputs to 0
    banksel OUTPORT
    movlw  b'00000000'
    movwf OUTPORT

    ; clear status pin
    bcf OUTPORT, PIN_STATUS

    ; rising edge on trigger pin
    bcf OUTPORT, PIN_TRIGGER
    bsf OUTPORT, PIN_TRIGGER

    ; this loop is to be left using faults
loop1
    bsf OUTPORT, PIN_STATUS
    goto loop1

    ; second loop to catch successful exit from loop1
loop2
    bcf OUTPORT, PIN_STATUS

    ; toggle pin
    bcf OUTPORT, PIN_STATUS_2
    nop
    bsf OUTPORT, PIN_STATUS_2
    nop
    goto loop2

    goto main
```

Our attack targets the `goto` instruction at the end of the first loop. Without external influence, the DUT never exits this loop. The aim of the fault attack is to jump over this instruction, so that the second loop gets executed. This condition can be detected by checking for `PIN_STATUS = 0`, indicating a successful fault injection. To provide a second indicator that the DUT is definitely executing the second loop, the toggling of `PIN_STATUS_2` can be tested. Note that a `goto` instruction

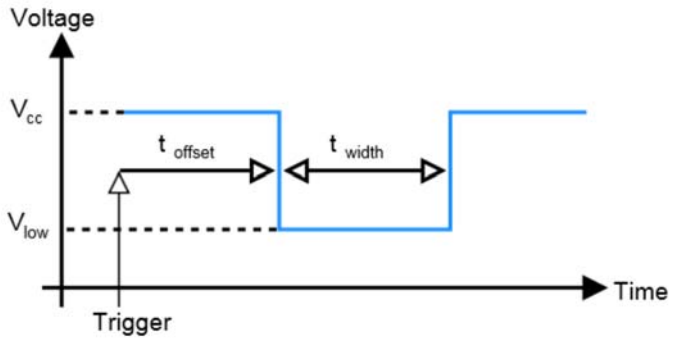


Figure 4.15.: Parameters characterising a single negative power glitch.

We investigate the use of a negative voltage glitch, as this method has been reported to be successful for other microcontrollers [KQ07, SH08]. In our framework, the following parameters (cf. Fig. 4.15) can be varied:

- The glitch offset t_{offset} with respect to the trigger rising edge,
- the glitch width t_{width} and
- the glitch voltage level V_{low}, i.e., the value the supply voltage is reduced to temporarily.

In order to systematically determine the settings that lead to the desired effect, we implemented an application with a "sweep mode" that consecutively tests all combinations in a certain range for each of the values. This way, the device is fully profiled for all possible parameters without human interaction. To demonstrate the integration of fault injection with the measurement framework and to be able to analyse the fault effect afterwards, the application also records oscilloscope traces of the voltage glitch on the supply rail and the state of the status pins.

Results Using the data gathered by the parameter sweep, three possible outcomes can be observed:

1. Injection not successful, i.e., `PIN_STATUS` remains set and the first loop is not left, see Fig. 4.17.

2. The device is reset, resulting in `PIN_STATUS` to be set low for a short time (during the initialization instructions) and then high again when the first loop is entered, see Fig. 4.16.
3. The desired fault is injected, i.e., `PIN_STATUS` stays low permanently, indicating that the microcontroller executes the second loop, see Fig. 4.18. Tab. 4.1 gives a selection of parameters[10] that lead to the desired fault effect.

V_{low} (V)	t_{offset} (ns)	t_{width} (ns)
1.65	1230	2210
1.65	2730	2210
	...	
1.73	2430	1810
1.73	3930	1810

Table 4.1.: Selection of successful single fault injection parameters.

Figures 4.17, 4.16 and 4.18 display example oscilloscope traces for each outcome. For the case 3, Fig. 4.19 additionally shows the toggling waveform on `PIN_STATUS_2` in `loop2` after a successful instruction skip fault.

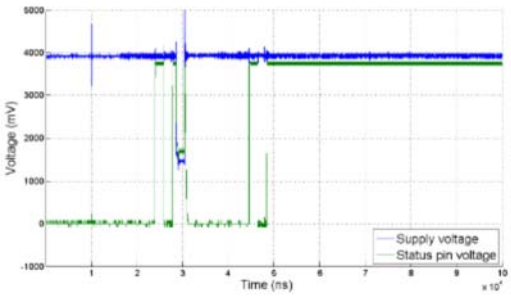


Figure 4.16.: Waveform of reset after fault injection on PIC16F687.

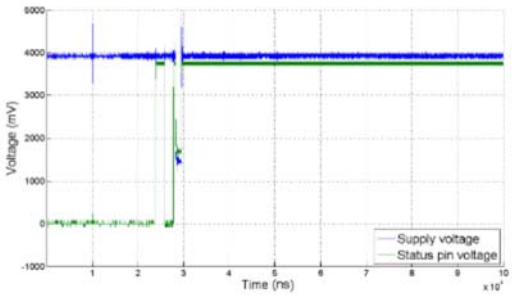


Figure 4.17.: Waveform of unsuccessful fault injection on PIC16F687.

Based on these experiments, we conclude that power glitch attacks to skip instruction on the PIC16F687 are possible. We identify three basic conditions for a successful fault injection:

Glitch voltage The reduced voltage V_{low} has to be within a region from 1.65 V to 1.73 V. For $V_{low} < 1.65$ V, the microcontroller is always reset. If $V_{low} > 1.73$ V, the power glitch does not affect the operation of the device.

[10] Note that there are many other settings that also function properly but have been left out for readability

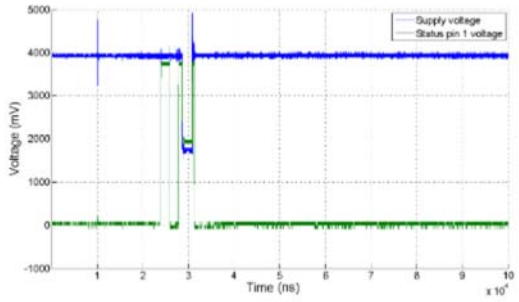


Figure 4.18.: Waveform of successful fault injection on PIC16F687, `PIN_STATUS`.

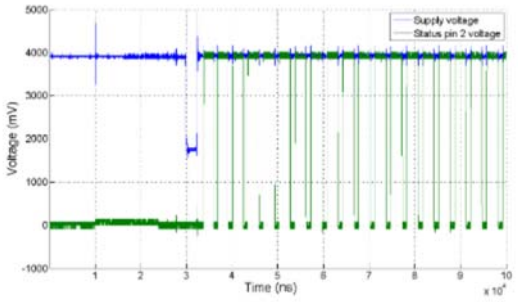


Figure 4.19.: Waveform of successful fault injection on PIC16F687, `PIN_STATUS_2`.

Glitch offset The offset t_{offset} must be set such that the *rising edge* of the glitch (i.e., the point after which the device resumes normal operation) matches the end of one `goto` instruction cycle. Hence, it is, e.g., possible to exit the loop with $t_{offset} \approx 2\,\mu$s, or in the subsequent iteration with $t_{offset} \approx 3.5\,\mu$s [11].

Glitch width The width of the glitch t_{width} is at least one loop iteration, i.e., $t_{width} > 1.5\,\mu$s. However, for certain voltage and offset combinations, the glitch may also be wider, again given that the rising edge appears at the end of one `goto` instruction cycle.

4.2.2. Multiple Faults

On the basis of the results of the previous section, the scenario is now extended to multiple fault injection. The first loop of the test code remains unchanged, while in the second loop the toggling of `PIN_STATUS_2` has been removed. The third loop catches the successful exit from `loop1` and `loop2`, indicating this condition by setting `PIN_STATUS_3` and toggling `PIN_STATUS_2` as an additional criterion for `loop3`.

```
main
    banksel OUTTRIS
    movlw  b'00000000'
    ; all pins are outputs
    movwf OUTTRIS
    ; set all outputs to 0
    banksel OUTPORT
    movlw  b'00000000'
    movwf OUTPORT
```

[11] The additional 0.5 μs result from the `bsf` instruction in the loop body, thus, one complete iteration (`bsf` + `goto`) takes 1.5 μs

```
    bcf OUTPORT, PIN_STATUS

    ; rising edge on trigger pin
    bcf OUTPORT, PIN_TRIGGER
    bsf OUTPORT, PIN_TRIGGER
loop1
    bsf OUTPORT, PIN_STATUS
    goto loop1

    ; clear first status pin
    bcf OUTPORT, PIN_STATUS
    nop
    nop

    ; second loop to catch successful exit from loop1
loop2
    bsf OUTPORT, PIN_STATUS_2
    goto loop2

    ; third loop to catch successful exit from loop2
loop3
    bsf OUTPORT, PIN_STATUS_3
    ; toggle pin
    bcf OUTPORT, PIN_STATUS_2
    nop
    bsf OUTPORT, PIN_STATUS_2
    nop
    goto loop3

    goto main
```

Thus, if the first two loops can be skipped using two successive faults, this condition is detected by checking for `PIN_STATUS = 0` and `PIN_STATUS_3 = 1`, which can again be accomplished automatically using the FPGA user I/O pins. For illustration, we recorded the waveform on `PIN_STATUS_2`, as the toggling provides visual evidence that the microcontroller is indeed executing `loop3`. The two successive negative voltage glitches are now characterised by 6 parameters, summarised in Fig. 4.20.

- The first glitch offset $t_{offset,1}$ with respect to the trigger rising edge,
- the first glitch width $t_{width,1}$,
- the first glitch voltage level $V_{low,1}$,

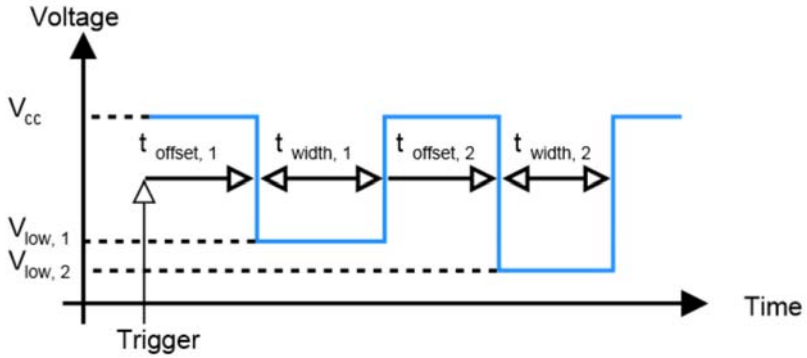


Figure 4.20.: Parameters characterising a double negative power glitch.

- the second glitch offset $t_{offset,2}$ with respect to the end of the first glitch,
- the second glitch width $t_{width,2}$ and
- the second glitch voltage level $V_{low,2}$.

To reduce the overhead for the search through all parameter combinations, the first glitch is fixed based on a setting that led to a successful fault in the single fault scenario. Moreover, the low voltage level is set equal for both glitches.

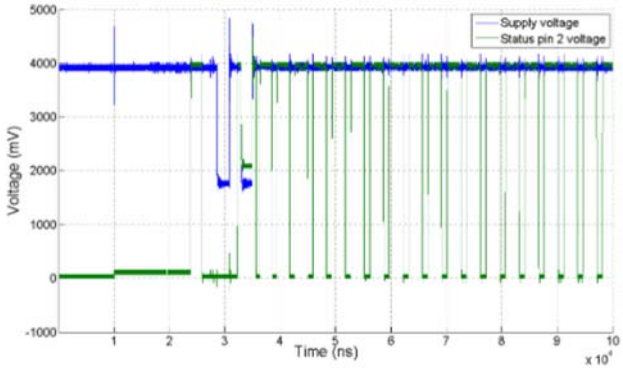


Figure 4.21.: Waveform of successful multiple fault injection on PIC16F687, `PIN_STATUS_2`.

Results By conducting a parameter sweep with the first pulse fixed, we were able to skip both `loop1` and afterwards `loop2`, resulting in the waveform in Fig. 4.21. Having profiled the device, we could repeat the experiments with identical parameters and reliably perform the fault injection, thereby achieving a success rate close to 100 %.

Table 4.2 provides settings that led to a successful multiple fault injection. Note that we limited the sweep parameter range as stated above in order to reduce the search

$V_{low,1} = V_{low,2}$ (V)	$t_{offset,1}$ (ns)	$t_{width,1}$ (ns)	$t_{offset,2}$ (ns)	$t_{width,2}$ (ns)
1.65	1230	2210	1930	2210
1.65	1230	2210	2130	2010
1.67	1230	2210	1730	2410
1.67	1230	2210	2130	2010

Table 4.2.: Selection of successful double fault injection parameters.

overhead. Generally, it can be assumed that there are far more settings that produce similar results.

Since all tests can be carried out automatically with varying parameters, and the success of the fault injection is automatically detected (the feedback via the data acquisition module even allows to determine, e.g., whether a reset of the DUT needs to be triggered), no human interaction is required. Thus, it is conceivable to perform a thorough profiling and find the correct points in time for inducing faults even in a black-box scenario.

The results presented in this section show that voltage faults allow to skip single and multiple instructions on a PIC microcontroller and thereby enable the modification of the normal program flow. Thus, many of the attacks given in Chap. 2 can be realised, including combined implementation attacks, e.g., by skipping the initialisation of a mask and performing a subsequent side-channel analysis. Due to the flexibility of the developed framework, it is possible to automatically determine suitable parameters for attacking a given implementation of a cryptographic algorithm, develop appropriate countermeasures against fault injection attacks, and finally estimate their effectiveness.

5. Conclusion

We conclude this thesis by summarising the achievements of our work, putting special emphasis on the flexibility and cost-effectiveness of the developed measurement and fault injection framework. Finally, we point out future research directions that may base on our setup and extend the practical results given in Chap. 4.

5.1. Summary

We give an overview of side-channel analysis and fault injection, providing a systematic list for the latter type of attack. By following a structured approach, we simplify the estimation whether a given device may be vulnerable to fault injection and which requirements have to be fulfilled for mounting a specific attack.

We present an adaptable and extensible framework for passive and active implementation attacks, not requiring expensive lab equipment: the complete setup can be built for approx. 2500 $, with a maximum measurement sample rate of 1 GSample/s and a fault injection precision of 10 ns. The framework can be used to put many theoretical approaches into practice. The modular approach allows for quick realisation of new techniques, particularly regarding different fault injection methods. As a proof-of-concept, we provide two modules designed to generate voltage glitches and clock faults.

To verify the effectiveness of the framework, we conduct side-channel analyses on a microcontroller-based AES software implementation and a commercial, widespread contactless smartcard employing a 3DES hardware engine. In both cases, we were able to recover the secret key and demonstrate the susceptibility to DPA and DEMA, respectively.

Previous work suggests that the injection of multiple faults is difficult to realise with low-cost equipment. Hence, countermeasures often only consider one fault during a computation. We counter this common belief by describing our method to skip multiple instructions on a popular microcontroller, using a setup that can be built by anyone with sufficient knowledge.

All of our work is in the public domain, so that the practical evaluation of theoretical attacks is simplified. The developed framework supports many different attacks proposed in the literature and can furthermore be extended for future scenarios. We expect it to

form the basis for new approaches in the field of implementation attacks, and, in this regard, give some ideas in the following section to conclude this thesis.

5.2. Future Work

First of all, we plan to add new extension boards to support fault injection techniques other than power glitches and variations of the clock. Currently, we develop appropriate PCBs for EM sparks and optical effects, keeping the cost-efficiency in mind. Besides, improvements of the fault injection hardware may also include the use of a different FPGA to further increase the timing precision, or the incorporation of a *Analogue-Digital Converter* (ADC) to enable dynamic triggering of the fault, depending on, e.g., a certain pattern in the power consumption of the DUT.

Besides, more practical results for fault attacks on real-world devices are needed to overcome the lack of publicly available information in this regard. For RFID tags, where the power is often supplied without direct contact, such attacks have been reported to be effective at least for very constrained devices [HSP08]. It would be interesting to see whether similar approaches are feasible in the case of contactless smartcards.

Finally, the combination of passive and active methods is an upcoming field of research which renders many traditional countermeasures against either form of attack ineffective. To be able to verify the effectiveness of new protection schemes and come up with solid results, practical experiments have to be conducted, for which our framework provides virtually all required capabilities.

A. Algorithms

Algorithm 6 Bellcore fault attack on RSA with CRT.

Require: Public parameters e, n known

Ensure: Recover private key $d = e^{-1} \mod \varphi(n)$

1: Request signature on x and inject fault during first exponentiation: $y \leftarrow [c_p q]\, y'_p + [c_q p]\, y_q \mod n$
2: Request signature on x without fault injection: $y \leftarrow [c_p q]\, y_p + [c_q p]\, y_q \mod n$
3: Compute $y - y' = [c_p q]\left(y_p - y'_p\right) \mod n$
4: Compute $q \leftarrow gcd\left(\left(y - y'\right), n\right) = gcd\left(\left[c_p q\right]\left(y_p - y'_p\right), pq\right)$
5: $p \leftarrow \frac{n}{q}$
6: $d \leftarrow e^{-1} \mod (p-1)(q-1)$
7: **return** d

Algorithm 7 Lenstra fault attack on RSA with CRT.

Require: Public parameters e, n known

Ensure: Recover private key $d = e^{-1} \mod \varphi(n)$

1: Request signature on x and inject fault during first exponentiation: $y \leftarrow [c_p q]\, y'_p + [c_q p]\, y_q \mod n$
2: **Note:** $(y')^e = [c_q p]\, y_q = x \mod q \Rightarrow (y')^e = \beta q + x$ for some $\beta \in \mathbb{Z}$
3: Compute $(y')^e - x = \beta q$
4: Compute $q \leftarrow gcd\left((y')^e - x, n\right) = gcd\left(\beta q, pq\right)$
5: $p \leftarrow \frac{n}{q}$
6: $d \leftarrow e^{-1} \mod (p-1)(q-1)$
7: **return** d

Algorithm 8 Fault attack on RSA without CRT.

Require: Public parameter n known

Require: Length l of binary representation of d known

Ensure: Recover one bit d_i of private key $d = e^{-1} \mod \varphi(n)$

1: Request valid signature on x: $y \leftarrow x^d \mod n$
2: Request signature on x and inject specified fault, targeting one bit of d : $y' \leftarrow x^{d'} \mod n$
3: Compute $c \leftarrow y' \cdot y^{-1} \mod n$
4: **for** $i \leftarrow 0 \ldots l-1$ **do** // *Candidates for fault position*
5: **if** $c == x^{2^i} \mod n$ **then**
6: $d_i \leftarrow 0$
7: **else if** $c == x^{-2^i} \mod n$ **then**
8: $d_i \leftarrow 1$
9: **else**
10: Fault did not occur at position i
11: **end if**
12: **end for**
13: **return** d_i

Algorithm 9 Fault analysis of EC multiplication with faulty input point.

Require: Coefficients of E: a_1, a_2, a_3, a_4, $a_6 \in \mathbb{F}$

Ensure: Recover the private key $d \in \mathbb{Z}$

while Equations for d do not allow for reconstruction using the CRT **do**
  Choose point P on E randomly
  Get ciphertext for P and inject a single-bit flip fault in the input P' of the multiplication
  Note: If P' is not on E, the device outputs $C' = (x'_C, y'_C) \leftarrow d \otimes P'$
  Determine a'_6 such that that C' satifies the curve equation:

$$a'_6 \leftarrow {y'}_C^2 + a_1 x'_C y'_C + a_3 y'_C - {x'}_C^3 - a_2 {x'}_C^2 - a_4 x'_C$$

  if E' defined by a_1, a_2, a_3, a_4, a'_6 is a non-singular EC and E' contains a small subgroup of order r **then** // *Check if DL problem has equivalent on weak curve* E'
    for all possible candidates P' with a single-bit fault **do**
      if P' lies on E' **then**
        Solve DL problem in the small subgroup: Given $\frac{ord(E')}{r} P'$, $\frac{ord(E')}{r} C' = \frac{ord(E')}{r} \cdot d \cdot P'$, compute $d_r \equiv d \mod r$.
        Add $d_r \equiv d \mod r$ to the list of equations
      end if
    end for
  end if
end while
return d

Algorithm 10 Fault analysis of EC with faults during the multiplication.

Require: Coefficients of E: a_1, a_2, a_3, a_4, $a_6 \in \mathbb{F}$

Require: Binary length n of d

Ensure: Set of candidates D for the most significant bits $d_n \dots d_{n-i}$ of the private key d

List of candidates $\mathcal{S} \leftarrow \emptyset$
Choose a point P on E
Get valid ciphertext for P: $Q_n \leftarrow d \cdot P$
Get ciphertext for P and inject a single-bit flip, non-permanent fault in register Q within the last m iterations: $Q'_n \leftarrow d \cdot P$
Assume: Fault affected register Q_j with $n - m \leq j < n$
Note: The actual value of j is not required
for $i \leftarrow n - m \dots n - 1$ **do** // *Candidate for first iteration $i > j$ with $d_i = 1$*
  for all $x \in \{0, 1\}^{n-i}$ **do** // *Candidate for i most significant bits of d*
    Candidate for $Q_i^{(x)}$ given i and x: $Q_i^{(x)} \leftarrow Q_n - x \cdot 2^i \cdot P$
    for all $Q'^{(x)}_i$ that result from a single-bit fault in $Q_i^{(x)}$ **do** // *Candidate for faulty intermediate Q'*
      Use the guess x for the i most significant bits to compute $Q'^{(x)}_n$, i.e., the faulty ouput given x and i
      if $Q'^{(x)}_n == {Q'}_n$ **then** // *Check simulated output*
        Add x as candidate for the i most significant bits of d: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(i, x)\}$
      end if
    end for
  end for
end for
return $\mathcal{D}$

Algorithm 11 Differential fault analysis of final round of DES.

Require: DES permutations $IP(\cdot)$, $Exp(\cdot)$, $P^{-1}(\cdot)$ and the S-Box layer $S(\cdot)$ formed by S-Boxes $S_1 \dots S_8$, message $x \in \{0, 1\}^{64}$

Ensure: The set of possible subkey candidates $\mathcal{K}$ for the S-Box affected by the single-bit fault is reduced (i.e., $|\mathcal{K}| < 2^6$) and contains the correct subkey K_{16} if algorithm did not output **Failure**

Encrypt message $x \in \{0, 1\}^{64}$: $y \leftarrow DES(x)$
Encrypt message $x \in \{0, 1\}^{64}$ and inject specified fault: $y' \leftarrow DES'(x)$
$(L_{16}, R_{16}) \leftarrow IP(y)$
$(L'_{16}, R'_{16}) \leftarrow IP(y')$
Compute output differential and undo permutation: $\Delta_{out} \leftarrow P^{-1}(R_{16} \oplus R'_{16})$
Note: $R_{16} \oplus R'_{16} = L_{15} \oplus f(R_{15}, K_{16}) \oplus L_{15} \oplus f'(R_{15}, K_{16}) = f(R_{15}, K_{16}) \oplus f'(R_{15}, K_{16})$
Find S-Box index i affected by the fault by observing positions of 1s in Δ_{out}
if $\Delta_{out} == 000 \dots 000$ **then** // *Check for zero differential*
  return Failure
end if
Subkey candidates for S-Box i: $\mathcal{K} \leftarrow \{\}$
Get 6 bit inputs before key XOR for S-Box i:
$in \leftarrow [Exp(L_{16})]_{\text{bit } (i-1)*6 \dots (i-1)*6+5}$
$in' \leftarrow [Exp(L'_{16})]_{\text{bit } (i-1)*6 \dots (i-1)*6+5}$
for all 6 bit subkey candidates K_{cand} for S-Box i **do**
  Get S-Box outputs for this candidate:
  $out \leftarrow S_i(in \oplus K_{cand})$
  $out' \leftarrow S_i(in' \oplus K_{cand})$
  Get output differential: $\Delta_{cand} = out \oplus out'$
  if $\Delta_{cand} == [\Delta_{out}]_{\text{bit } (i-1)*4 \dots (i-1)*4+3}$ **then** // *Check if candidate yields valid output differential*
    $\mathcal{K} \leftarrow \mathcal{K} \cup \{K_{cand}\}$
  end if
end for
return $\mathcal{K}$

Algorithm 12 Differential fault analysis of final round of AES.

Require: AES operations $SubBytes(\cdot)$ and $ShiftRows^{-1}(\cdot)$

Require: Function $ShiftRowPos(j)$ yielding the position of byte j after $ShiftRows(\cdot)$

Ensure: Recover the 128 bit subkey K_{10} byte-wise, whereas a single byte is obtained with 97% success probability using 3 faulty ciphertexts. For the complete 16 byte key, ≈ 50 faulty ciphertexts are thus sufficient with high probability.

1: Get valid ciphertext: $C \leftarrow AES128(x)$
2: **for all** byte $j = 0 \ldots 15$ **do** // *Obtain M_9 byte-wise*
3: Set of candidates for byte j of M_9: $\mathcal{M}^j_{cand} \leftarrow \{0 \ldots 255\}$
4: Get permuted position after final $ShiftRows(\cdot)$: $j' \leftarrow ShiftRowPos(j)$
5: **while** $\left|\mathcal{M}^j_{cand}\right| \neq 1$ **do** // *Reduce candidate set for M_9*
6: Get ciphertext and inject single-bit fault at byte j: $C' \leftarrow AES128'(x)$
7: Output differential: $\Delta_{out} \leftarrow C \oplus C'$
8: **if** $[\Delta_{out}]_{\text{byte } j'} == 0$ **then** // *Check if fault occured at correct byte*
9: No fault at byte j: **Continue** on line 5
10: **else**
11: Set of candidates for byte j of M_9 for this differential: $\mathcal{M}^j_{\Delta} \leftarrow \{\}$
12: **for all** possible single-bit faults $e_j \in \{0000001, 00000010, \ldots, 10000000\}$ **do**
13: Get all candidates $\hat{M}_9$ that fulfil

$$[\Delta_{out}]_{\text{byte } j'} = \left[SubBytes\left(\left[\hat{M}_9\right]_{\text{byte } j}\right) \oplus SubBytes\left(\left[\hat{M}_9 \oplus e_j\right]_{\text{byte } j}\right)\right]_{\text{byte } j'}$$

and store in set $\mathcal{M}^j_{\Delta, e_j}$
14: Update candidates: $\mathcal{M}^j_{\Delta} \leftarrow \mathcal{M}^j_{\Delta} \cup \mathcal{M}^j_{\Delta, e_j}$
15: **end for**
16: Reduce set of overall candidates: $\mathcal{M}^j_{cand} \leftarrow \mathcal{M}^j_{cand} \cap \mathcal{M}^j_{\Delta}$
17: **end if**
18: **end while**
19: $[M_9]_{\text{byte } j} \leftarrow$ only candidate left in $\mathcal{M}^j_{cand}$
20: **end for**
21: Recover subkey: $K_{10} \leftarrow C \oplus ShiftRows(SubBytes(M_9))$
22: **return** K_{10}

Algorithm 13 Fault analysis of pre-whitening step of AES.

Require: Plaintext is constant 0: $x \leftarrow 000\ldots000$

Ensure: Recover the 128 bit subkey K_0 bit-wise

1: Get valid ciphertext: $C \leftarrow AES128\,(x)$
2: **for** $l = 0, 1, \ldots, 127$ **do** // *Scan through bits of* K_0
3: Get ciphertext and inject fault, setting bit l of M_0 to 0: $C' \leftarrow AES128'\,(x)$
4: **if** $C' == C$ **then** // *bit* l *of* K_0 *was 0* $\Rightarrow$ *fault had no impact*
5: $[K_0]_{\text{bit } l} \leftarrow 0$
6: **else** // *bit* l *of* K_0 *was 1* $\Rightarrow$ *fault changed ciphertext*
7: $[K_0]_{\text{bit } l} \leftarrow 1$
8: **end if**
9: **end for**
10: **return** K_0

B. Framework Documentation

The information in this appendix is a summary of the *Doxygen* source-level documentation for the framework. The complete documentation is provided in HTML format along with the source code.

B.1. Class List

The following list contains the name and a short description for all relevant framework classes. For more details, refer to the actual HTML documentation.

alignment Alignment processing block

alignment_visitor Wrapper visitor class for state updates in processing chain

analysis_config Analysis configuration Storage for analysis settings

analysis_parameters

app Measurement app example: Sample class to demonstrate inheritance from measurement_app

bandpass Bandpass processing block

binwise_correlation_dpa< TAlgorithm > Manager class for several key hypotheses in a binwise manner

binwise_dpa_candidate< TAlgorithm > Correlation coefficient for binwise DPA

bytewise_distance< TResult > Bytewise hamming distance

channel_config Channel settings Storage class for channel specific settings

comb Comb filter processing block

configurable Interface for configuration sub-objects

correlation_dpa< TAlgorithm > Manager class for several key hypotheses

cut_trace Cut trace block Cut part of trace

dbgbuf Debug output stream

dbgstream Debug stream

default_trace_load_functor Default trace loader: Standard binary trace loader using first enabled channel in measurement config file

demodulator 13.56 MHz Demodulator and decoder

des_oracle DES oracle with intermediates before and after each round

downsampler Downsampling by integer factor

dpa_app DPA App for CPA on 3DES smartcard

dpa_candidate< TAlgorithm > Correlation coefficient for standard pointwise DPA

dsp_trace_average Recursive timeseries average and variance estimator

dtw Dynamic time warping processing block

fault_config Fault fpga config Storage for fault FPGA config

fault_fpga Fault injection VHDL control class

fft FFT wrapper class

fir_filter FFT-based FIR filter

hamming_distance< TResult > Standard hamming distance model

hamming_weight< TResult > Hamming weight model

iir_hp IIR highpass (dc blocker) processing block

job Abstract base class for processing job

job_manager Class holding processing jobs

job_state Storage for job state attributes

keyboard Non-blocking keyboard access

measurement_app Measurement framework: Framework class to inherit measurement app from, provides useful basic functions for recording traces and injecting faults

measurement_config Config class for recording

mifare_classic Class for Ultralight C commands

observable Base class for observable objects

observer Abstract base class for observers

PBase< T > Base class for type wrapping

peak_dpa_config Configuration for peak extraction DPA: Storage for peak extraction settings

peak_extract Peak extraction app Peak extraction processing application

peak_load_functor Trace loader for peak DPA: Load peak values and position from ascii file (as time value pairs) with optional prefiltering (only use minima/maxima)

picoscope Picoscope interface class

power_model< TResult > Abstract base class for power models

processing_app Trace processing framework class: Framework class to inherit processing/evaluation app from, provides useful basic functions for processing traces and DPA like attacks

processing_chain Container class for data processors

processing_config Configuration for processing Storage for analysis settings

psd Power spectral density Estimate psd using FFT

reader_config RFID reader config

recording_config Config object for recording settings

rectifier Rectifier processing block

rfid_device RFID device base class

scope Abstract base class for all scopes

scope_config Scope settings: Storage class for scope specific settings

second_order Second order DPA pre-processing block

signed_distance< TResult > Signed distance model

trace Base class for trace file handling

trace_agilent_bin Agilent BinFormat trace class

trace_ascii ASCII double values trace class

trace_binary Binary format trace loading

trace_display_box Box widget to draw a trace onto

trace_display_window Class to display trace Bases on fltk, which is available as cygwin package

trace_display_window_manager Manager class for trace windows Bases on fltk, which is available as cygwin package

trace_format_parameters Trace format storage class

trace_load_functor Abstract base class for functor for trace loading

trace_processor Abstract base class for in-place operator on timeseries data

trace_processor_visitable Interface for visitable processing chain elements

trace_processor_visitor Interface for visiting processing chain elements

trigger_channel_properties Settings for one channel: Storage class for properties

trigger_config Trigger settings Storage class for trigger settings

tripledes_dpa_config Configuration for 3DES DPA: Storage for DPA settings

tripledes_oracle 3DES oracle with values before and after each Single-DES

util Collection class for several utility functions

value_locked< T > Locked thread-safe value wrapper

windowed_psd Windowed power spectral density Spectral density based on (overlapped) windowed frames

B.2. Class Hierachy

As the framework is heavily employing inheritance, we give an overview over the inheritance structure of the above classes. Again, more information is available in the complete *Doxygen* HTML documentation.

- analysis_config
- analysis_parameters
- binwise_correlation_dpa< TAlgorithm >
- binwise_dpa_candidate< TAlgorithm >
- configurable
  - channel_config
  - fault_config
  - peak_dpa_config
  - reader_config
  - recording_config
  - scope_config
  - trigger_config
  - tripledes_dpa_config
- correlation_dpa< TAlgorithm >
- dbgbuf
- des_oracle_base
  - des_oracle
  - tripledes_oracle
- dpa_candidate< TAlgorithm >
- dsp_trace_average

- fault_fpga
- fft
- fir_filter
- job_state
- keyboard
- measurement_app
  - app
  - demodulator
- measurement_config
- observable
  - job
  - job_manager
- observer
  - job_manager
- PBase< T >
- PBase< dbgbuf >
  - dbgstream
- power_model< TResult >
- power_model< TResult >
  - bytewise_distance< TResult >
  - hamming_distance< TResult >
  - hamming_weight< TResult >
  - signed_distance< TResult >
- processing_app
  - dpa_app
  - peak_extract
- processing_config
- rfid_device
- scope
  - picoscope
- trace

    - trace_agilent_bin
    - trace_ascii
- trace_binary
- trace_display_box
- trace_display_window
- trace_display_window_manager
- trace_format_parameters
- trace_load_functor
    - default_trace_load_functor
    - peak_load_functor
- trace_processor_visitable
    - processing_chain
    - trace_processor
        * alignment
        * bandpass
        * comb
        * cut_trace
        * downsampler
        * dtw
        * iir_hp
        * psd
        * rectifier
        * second_order
        * windowed_psd
- trace_processor_visitor
    - alignment_visitor
    - alignment_visitor
    - alignment_visitor
- trigger_channel_properties
- util
- value_locked< T >

C. Schematics

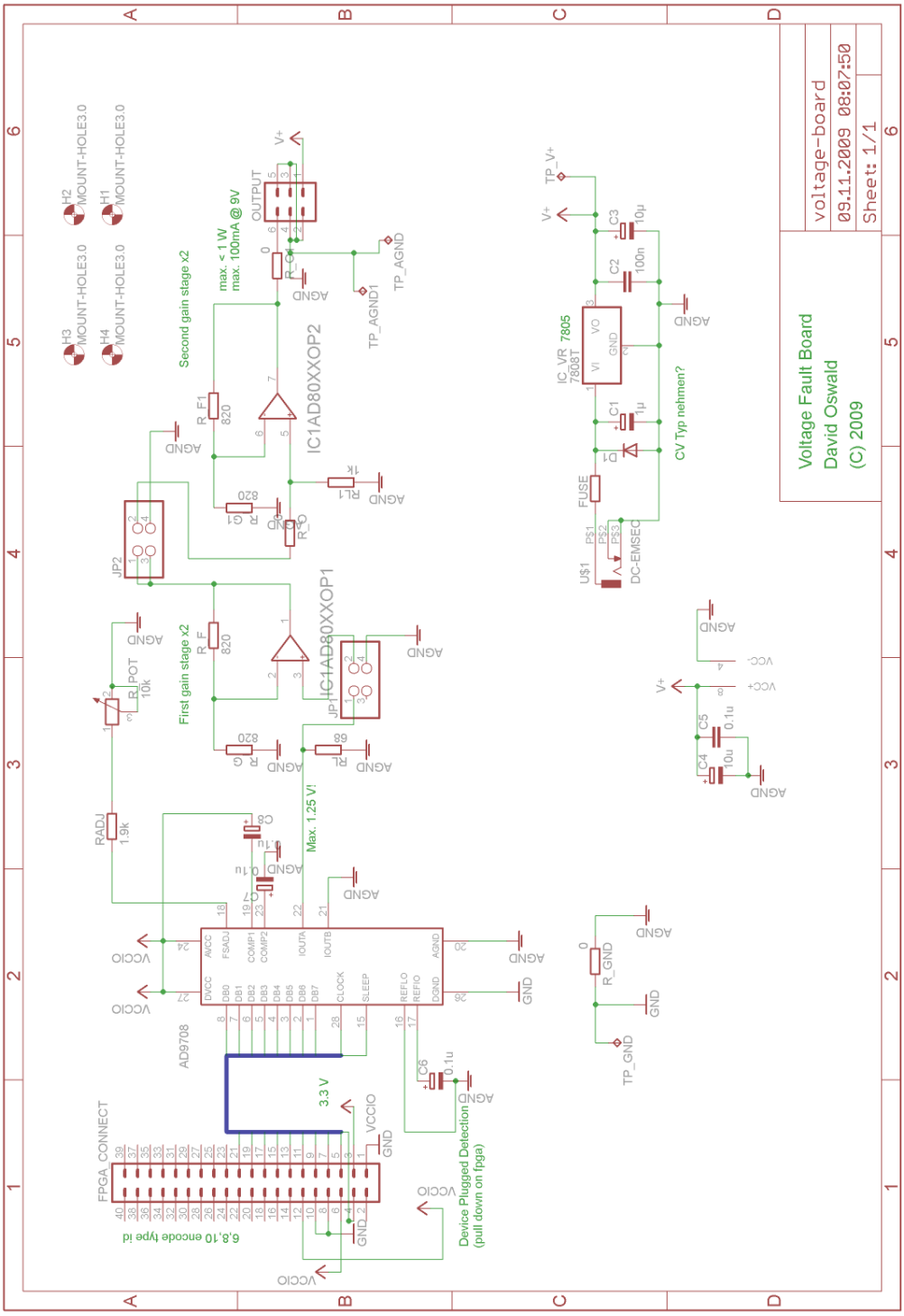


Figure C.1.: Schematic of power fault module, revision 1.

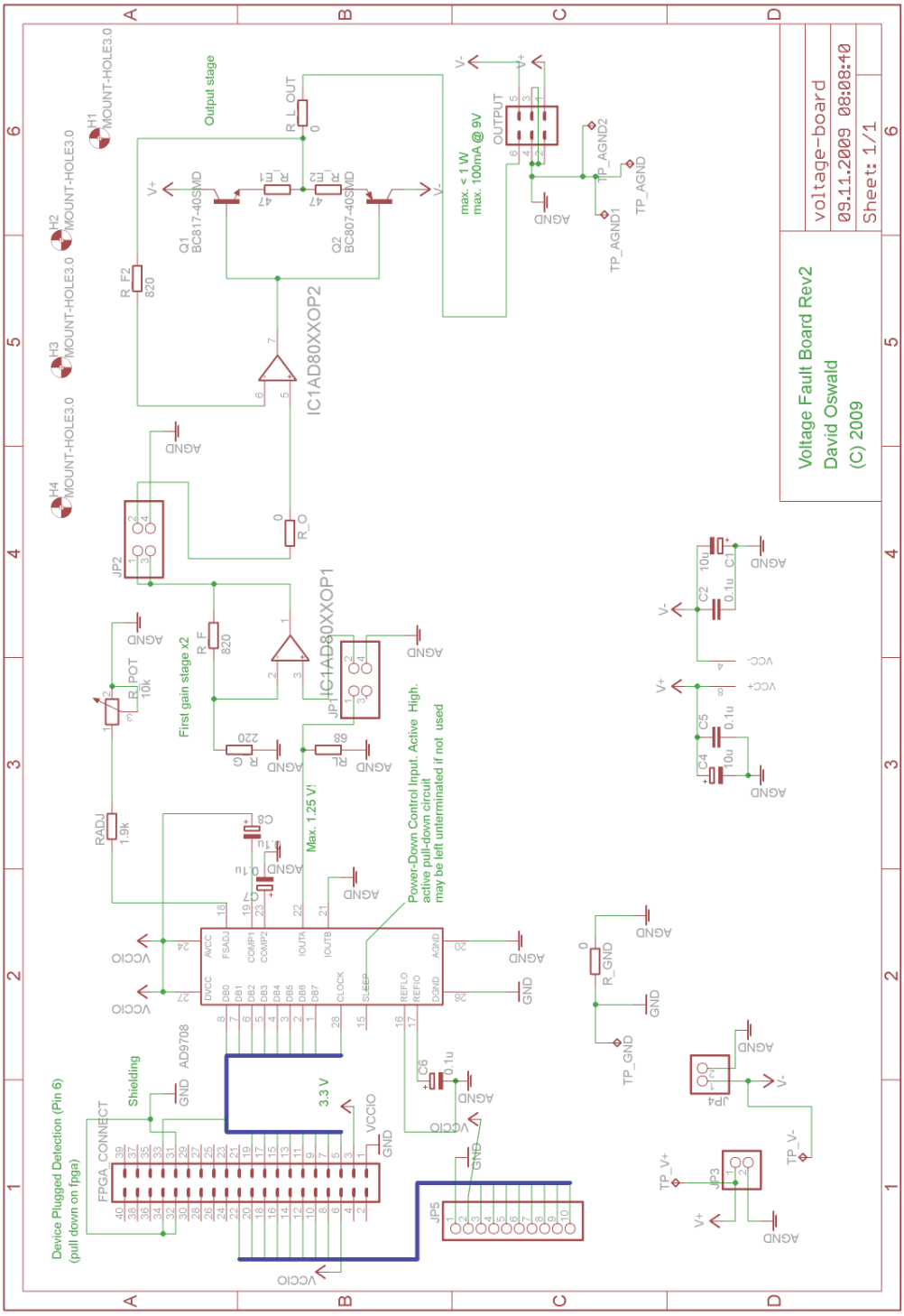


Figure C.2.: Schematic of power fault module, revision 2.

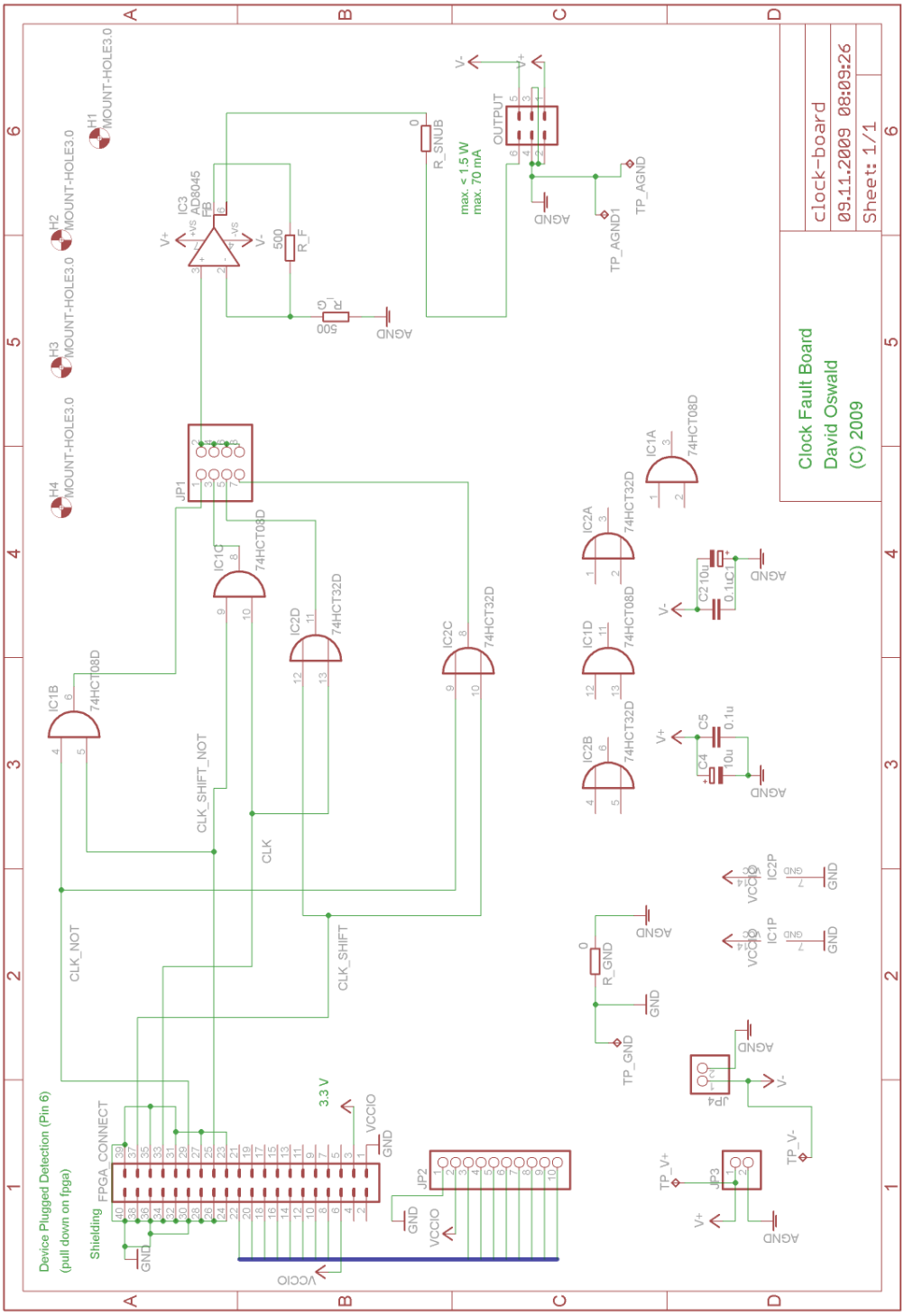


Figure C.3.: Schematic of clock fault module.

D. Bibliography

[Ana04] Analog Devices, Inc.: *AD8045 Voltage Feedback High Speed Amplifier Datasheet*, 2004. http://www.analog.com/static/imported-files/data_sheets/AD8045.pdf

[Ana09a] Analog Devices, Inc.: *AD8012 Dual Low Power Current Feedback Amplifier Datasheet*, 2009. http://www.analog.com/static/imported-files/data_sheets/AD8012.pdf

[Ana09b] Analog Devices, Inc.: *AD9708 8-Bit, 100 MSPS+ TxDAC D/A Converter Datasheet*, 2009. http://www.analog.com/static/imported-files/data_sheets/AD9708.pdf

[Atm03] Atmel, Inc.: *ATmega163 8-bit Microcontroller with 16K Bytes In-System Programmable Flash Datasheet*, 2003. http://www.atmel.com/dyn/resources/prod_documents/doc1142.pdf

[AVFM07] Amiel, Frederic ; Villegas, Karine ; Feix, Benoit ; Marcel, Louis: Passive and Active Combined Attacks: Combining Fault Attacks and Side Channel Analysis. In: *FDTC '07: Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography*. Washington, DC, USA : IEEE Computer Society, 2007, pages 92–102

[BCO04] Brier, Eric ; Clavier, Christophe ; Olivier, Francis: Correlation Power Analysis with a Leakage Model. In: Joye, Marc ; Quisquater, Jean-Jacques : *Cryptographic Hardware and Embedded Systems - CHES 2004* Bd. 3156, Springer, 2004 (Lecture Notes in Computer Science), pages 16–29

[BDH+97] Bao, F. ; Deng, R. H. ; Han, Y. ; A.Jeng ; Narasimhalu, A. D. ; Ngair, T.: Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults. In: *Proceedings of the 5th International Workshop on Security Protocols*, Springer-Verlag, 1997, pages 115–124

[BDL97] Boneh, Dan ; Demillo, Richard A. ; Lipton, Richard J.: On the Importance of Checking Computations. In: *Proceedings of Eurocrypt '97*, 1997, pages 37 – 51

[BMM00] Biehl, Ingrid ; Meyer, Bernd ; Müller, Volker: Differential Fault Attacks on Elliptic Curve Cryptosystems. In: *Advances in Cryptology - CRYPTO 2000*, Springer-Verlag, 2000, pages 131–146

[Boo] Boorman, Brian: *VHDL Code Implements 50 Percent-Duty-Cycle Divider*. EDN Magazine. http://www.rhinocerus.net/forum/lang-vhdl/103649-clock-divider.html

[BP03] Benso, Alfredo ; Prinetto, Paolo: *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Kluwer Academic Publishers, 2003

[Bre09] Break IC, Inc.: *Homepage*. http://www.break-ic.com/topics/break-ic.asp. version 2009. – web resource

[BS97] Biham, Eli ; Shamir, Adi: Differential Fault Analysis of Secret Key Cryptosystems. In: *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, 1997, pages pages 513 – 525

[BS03] Blömer, Johannes ; Seifert, Jean-Pierre: Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In: *Financial Cryptography'03*, Springer-Verlag, 2003, page 162-181

[BSI] BSI - German Ministry of Security: *ePass - Der Reisepass mit biometrischen Merkmalen*. http://www.bsi.de/fachthem/epass/,

[Car05] Carluccio, Dario: *Electromagnetic Side Channel Analysis for Embedded Crypto Devices*, Ruhr Universität Bochum, Diplomarbeit, 2005

[CFA+06] Cohen, Henri ; Frey, Gerhard ; Avanzi, Roberto ; Douce, Christophe ; Lange, Tanja ; Nguyen, Kim ; Vercautern, Frederik: *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2006

[CMCJ04] Chevallier-Mames, Benoît ; Ciet, Mathieu ; Joye, Marc: Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. In: *IEEE Transactions on Computers* 53 (2004), pages 760–768

[CNO08] Courtois, Nicolas T. ; Nohl, Karsten ; O'Neil, Sean: Algebraic Attacks on the Crypto-1 Stream Cipher in MiFare Classic and Oyster Cards. (2008). http://eprint.iacr.org/2008/166

[Cor05] Corson, D.W.: Comparing 8-bit Microcontrollers for Ultra-Low-Power Applications. (2005), page 3, table 1. http://www.embedded-control-europe.com/c_ece_knowhow/680/eceoct05p22.pdf

[Dev08] Devillard, Nicolas: *iniParser: stand-alone ini Parser library in ANSI C*. electronic resource. http://ndevilla.free.fr/iniparser/. version 2008

[EKM+08] Eisenbarth, Thomas ; Kasper, Timo ; Moradi, Amir ; Paar, Christof ; Salmasizadeh, Mahmoud ; Shalmani, Mohammad T. M.: On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoq Code Hopping Scheme. In: *Advances in Cryptology - CRYPTO 2008* Bd. 5157, Springer, 2008 (Lecture Notes in Computer Science), pages 203–220

[FIPa] *FIPS 197 Advanced Encryption Standard (AES)*. – http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[FIPb] *FIPS 46-3 Data Encryption Standard (DES).* – `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`

[FML+03] Fournier, Jacques J. A. ; Moore, Simon ; Li, Huiyun ; Mullins, Robert ; Taylor, George: Security Evaluation of Asynchronous Circuits. (2003), pages 137–151

[GCC09] GCC team: *GCC, the GNU Compiler Collection.* electronic resource. `http://gcc.gnu.org/`. version September 2009

[Gir03] Giraud, Christophe: DFA on AES. In: *Advanced Encryption Standard - AES, 4th International Conference, AES 2004*, Springer, 2003, pages 27–41

[GT04] Giraud, Christophe ; Thiebeauld, Hugues: A Survey on Fault Attacks. In: Quisquater, Jean-Jacques ; Paradinas, Pierre ; Deswarte, Yves ; Kalam, Anas Abou E. : *CARDIS*, Kluwer, 2004, pages 159–176

[HCN+04] Hamid, Hagai Bar-El ; Choukri, Hamid ; Naccache, David ; Tunstall, Michael ; Whelan, Claire: The Sorcerer's Apprentice Guide to Fault Attacks, 2004

[HSP08] Hutter, Michael ; Schmidt, Jörn-Marc ; Plos, Thomas: RFID and Its Vulnerability to Faults. In: Oswald, Elisabeth ; Rohatgi, Pankaj : *Cryptographic Hardware and Embedded Systems, CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008, Proceedings*, Springer, 2008 (Lecture Notes in Computer Science (LNCS)), pages 363 – 379

[iso01a] International Organization for Standardization, Geneva, Switzerland: *ISO/IEC 14443-3: Identification cards - Contactless integrated circuit(s) cards - Proximity cards - Part 3: Initialization and anticollision.* 1st. February 2001

[iso01b] International Organization for Standardization, Geneva, Switzerland: *ISO/IEC 14443-4: Identification cards - Contactless integrated circuit(s) cards - Proximity cards - Part 4: Transmission protocol.* 1st. February 2001

[ISO03] International Organization for Standardization, Geneva, Switzerland: *ISO 7810 Identification Cards - Physical Characteristics.* `www.iso.org`. version 2003

[ISO04] International Organization for Standardization, Geneva, Switzerland: *ISO 7816 Identification Cards - Integrated Circuit Cards with Contacts.* `www.iso.org`. version 2004

[Kas06] Kasper, Timo: *Embedded Security Analysis of RFID Devices*, Ruhr Universität Bochum, Diplomarbeit, 2006

[KJJ99] Kocher, Paul C. ; Jaffe, Joshua ; Jun, Benjamin: Differential Power Analysis. In: *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology.* London, UK : Springer-Verlag, 1999. – ISBN 3–540–66347–9, pages 388–397

[KK99] Kömmerling, Oliver ; Kuhn, Markus G.: Design Principles for Tamper-Resistant Smartcard Processors, 1999, pages 9–20

[KK06] Kammeyer, Karl-Dirk ; Kroschel, Kristian: *Digitale Signalverarbeitung.* 6th. Teubner Verlag, 2006

[KOP09] Kasper, Timo ; Oswald, David ; Paar, Christof: EM Side-Channel Attacks on Commercial Contactless Smartcards using Low-Cost Equipment. In: *Proceedings of WISA 2009*, to appear in Springer LNCS, September 2009

[KQ07] Kim, Chong H. ; Quisquater, Jean-Jacques: Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures. In: *WISTP*, 2007, pages 215–228

[Kra04] Krasner, Jerry: Using Elliptic Curve Cryptography (ECC) for Enhanced Embedded Security. (2004). `http://embeddedforecast.com/EMF-ECC-FINAL1204.pdf`

[Kug03] Kugelstadt, Thomas: *Op Amps for Everyone.* 2nd. Texas Instruments, 2003

[Lan] Langer EMV-Technik: *Details of Near Field Probe Set RF 2.* web resource. `http://www.langer-emv.de/en/produkte/prod_rf2.htm`

[Len96] Lenstra, Arjen: *Memo on RSA Signature Generation in the Presence of Faults.* 1996. – Manuscript

[Mic08] Microchip Technology Inc.: *PIC16F631/677/685/687/689/690 Data Sheet 20-Pin Flash-Based, 8-Bit CMOS Microcontrollers with nanoWatt Technology*, 2008. `http://ww1.microchip.com/downloads/en/DeviceDoc/41262E.pdf`

[MOP07] Mangard, Stefan ; Oswald, Elisabeth ; Popp, Thomas: *Power Analysis Attacks: Revealing the Secrets of Smart Cards.* Secaucus, NJ, USA : Springer-Verlag, 2007 `http://www.dpabook.org/`. – ISBN 978–0–387–30857–9

[Osw08] Oswald, David: *On the Feasibility of Differential Electromagnetic Analysis of Contactless Smartcards*, Ruhr Universität Bochum, Studienarbeit, 2008

[Paa06] Paar, Christof: *Implementierung Kryptographischer Verfahren I.* Lecture notes, 2006

[Pic07] Pico Technology: *PicoScope 5000 Series User's Guide*, November 2007. `http://www.picotech.com/document/pdf/ps5000-en.pdf`

[Pic08] Pico Technology: *PicoScope 5200 USB PC Oscilloscopes.* `http://www.picotech.com/picoscope5200-specifications.html`. version 2008

[Pot09] Potato Semiconductor Corporation: *PO74G08A Quadruple 2-Input Positive AND Gate*, July 2009. `http://www.potatosemi.com/datasheet/PO74G08A.pdf`

[Red09] Red Hat Cygwin team: *Cygwin: a Linux-like environment for Windows.* electronic resource. `http://www.cygwin.com/`. version September 2009

[Sch08] Schmidt, Jörn-Marc: Differential Fault Analysis - Final Report / TU Graz. 2008. – Forschungsbericht

[SH08] Schmidt, J.-M. ; Herbst, C.: A Practical Fault Attack on Square and Multiply. In: *Proc. 5th Workshop on Fault Diagnosis and Tolerance in Cryptography FDTC '08*, 2008, pages 53–58

[Sha06] Shanmugam, K. S.: *Digital & Analog Communication Systems.* Wiley-India, 2006. – ISBN 8126509147

[Sko01] Skorobogatov, Sergei P.: Copy Protection in Modern Microcontrollers. (2001). `http://www.cl.cam.ac.uk/~sps32/mcu_lock.html`

[TES] TESTEC Elektronik GmbH: *Passive Tastköpfe HF bis 350 MHz.* `http://www.testec.de/tastkoepfe_hf_350mhz0.0.html`

[Wik09] Wikipedia: *Advanced Encryption Standard — Wikipedia, Die freie Enzyklopädie.* `http://de.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=63118407`. version 2009. – [Online; Stand 25. August 2009]

[WW05] Waddle, Jason ; Wagner, David: Fault Attacks on Dual-Rail Encoded Systems. In: *Computer Security Applications Conference, Annual* (2005), pages 483–494. – ISSN 1063–9527

[Xil07] Xilinx, Inc.: *Virtex-4 Family Overview*, September 2007. `http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf`

[Xil08a] Xilinx, Inc.: *PicoBlaze 8-bit Embedded Microcontroller User Guide.* v. 1.1.2, June 2008. `http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf`

[Xil08b] Xilinx, Inc.: *Spartan-3 FPGA Family Data Sheet.* v 2.4, June 2008. `http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf`

[Xil08c] Xilinx, Inc.: *Spartan-3 FPGA Starter Kit Board User Guide.* v 1.2, June 2008. `http://www.xilinx.com/support/documentation/boards_and_kits/ug130.pdf`

[Xil09a] Xilinx, Inc.: *ISE WebPACK Design Software.* electronic resource. `http://www.xilinx.com/tools/webpack.htm`. version September 2009

[Xil09b] Xilinx, Inc.: *PicoBlaze User Resources* . web resource, 2009