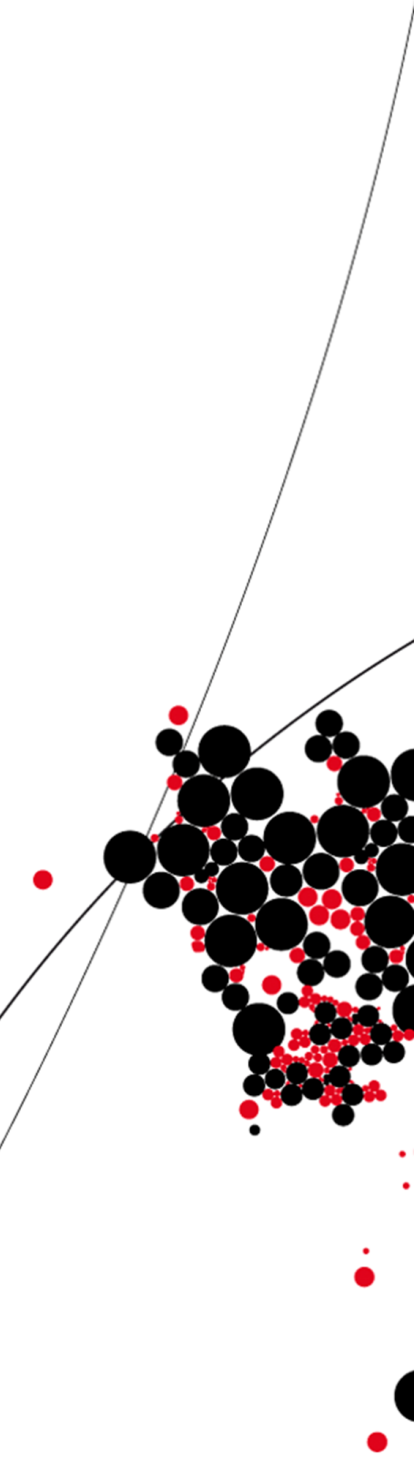
UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Developing a Platform to Emulate Fault Injection Attacks on Cryptographic Implementations

E. M. van der Ploeg

B.Sc. Thesis
January 2020



Supervisors:
dr. ing. D. M. Ziener
A. Asghar, M.Sc.
dr. C. G. Zeinstra

Computer Architecture
for Embedded Systems
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Field Programmable Gate Arrays (FPGAs) are used more and more in security-critical applications. With this development comes increased attention from both benevolent and malevolent actors, interested in tampering with the security of these cryptographic implementations. These actors take interest in the possibilities of cracking the security of these applications. When an encryption algorithm is implemented in hardware, there is a possibility of a Fault Injection Attack (FIA). Through these attacks, faulty ciphertexts that contain information on the cryptosystem's secret key can be obtained. Through Differential Fault Analysis (DFA), the secret key can be derived. This work aims to develop a platform to help designers of cryptographic implementations on FPGAs test their design against FIAs. By reconfiguring an FPGA with a modified configuration file, a FIA can be emulated. The configuration file (bitstream) has to be altered to make this possible. The EDA toolkit DAVOS, originally aimed at reliability analysis of FPGA implementations, is re-purposed for this particular purpose, as it has the FPGA re-programming functionality required for this purpose. Successful FIA emulations, with extraction of the cryptosystem's secret key, are demonstrated. DAVOS still needs to be optimized to be used effectively as an FIA emulator, thus suggestions are made to improve its use for this purpose.

Contents

1 Introduction **3**

2 The Advanced Encryption Standard (AES) **4**
2.1 AES 4
2.2 SubBytes 5
2.3 ShiftRows 6
2.4 MixColumns 7
2.5 AddRoundKey 7

3 Differential Fault Analysis (DFA) **9**
3.1 The Giraud Attack 9
3.2 Generalized attacks on AES 11

4 Fault Injection Attacks (FIAs) on FPGAs **14**
4.1 FPGA Architecture 14
4.2 Practical Feasibility 14

5 Countermeasures against FIAs **15**
5.1 Redundancy 15
5.2 Advanced Solutions 17

6 Bitstream-based FIA emulation using DAVOS **18**
6.1 Bitstream manipulation and reverse engineering 18
6.2 Existing FIA emulation 18
6.3 DAVOS as an FIA emulation tool 18
6.4 Methods 20
6.5 Emulating a Giraud attack 20
6.6 Attacking the input of the 9^{th} round 21
6.7 Attacking a TMR implementation with DAVOS 21

7 Evaluation **22**

8 Conclusions and Recommendations **23**

9 Bibliography **24**

A Key extraction and expansion **26**
A.1 Byte-wise key extraction used in the Giraud attack 26
A.2 Byte-by-byte key extraction used for attacking the 9^{th} round input 28
A.3 Inverse key expansion 33

B VHDL TMR Implementation **36**
B.1 Top-level TMR description 36
B.2 Voter 37

1 Introduction

In recent times the demand for both flexible, fast and cheap hardware solutions is ever increasing. FPGAs offer a good middle ground regarding these criteria. They are flexible due to their reprogrammability, offer better computation speeds than software solutions and don't require the massive upfront investment ASICs do. This has led to more and more systems making use of these reprogrammable hardware devices. Such systems often make use of cryptographic implementations, e.g. AES, to securely transmit information. With the increased use of FPGAs in security-critical applications, there is now an increased emphasis on securing these devices from tampering. One way to tamper with cryptographic implementations is a Fault Injection Attack (FIA). In such an attack, an adversary attempts to induce faults, e.g. single-bit flips. The adversary then compares the faulty outputs of the cryptographic implementation to the correct output, and attempts to retrieve the secret key using Differential Fault Analysis (DFA). FPGA developers aware of the potential security risks from FIAs will want to test the robustness of their design against these attacks. In this research, the EDA toolkit DAVOS, traditionally used for reliability analysis, will be used to emulate FIAs on cryptographic implementations on an FPGA. This could allow a user to execute a FIA against their design, or check how their countermeasures against these attacks hold up. In this thesis, focus is placed on the Advanced Encryption Standard, but DAVOS can in principle induce single-bit faults in any FPGA design, and thus in any cryptographic implementation. At the best of our knowledge, this is the first example of FIA emulation on a cryptographic system running on an FPGA.

The thesis is structured as follows:

- Chapter 2: The Advanced Encryption Standard (AES)
- Chapter 3: Differential Fault Analysis (DFA) on AES
- Chapter 4: Fault Injection Attacks (FIAs) on FPGAs
- Chapter 5: Countermeasures against FIAs
- Chapter 6: Bitstream-based FIA Emulation using DAVOS
- Chapter 7: Evaluation
- Chapter 8: Conclusions and Recommendations

2 The Advanced Encryption Standard (AES)

2.1 AES

This research will focus on successfully emulating FIAs on implementations of the Advanced Encryption Standard (AES). Published in 1998, this fixed block length implementation of the Rijndael cipher [1] was chosen by NIST to become the successor of the Data Encryption Standard (DES). The symmetric key encryption algorithm is famed for its simplicity, while still providing virtually uncrackable encryption.

In his 1945 work 'A mathematical theory of Cryptography' [2] for Bell labs, Shannon identified two important aspects of a strong cryptographic algorithm: Confusion and diffusion. Confusion makes the relationship between the key and the ciphertext as complex as possible. Diffusion, on the other hand, makes relationship between the plaintext and the ciphertext complex. Thus, an encryption algorithm with good diffusion and confusion should have its output completely changed, even if only one bit of the input or the key is flipped.

AES, like many other encryption algorithms, has good confusion and diffusion because it is a so-called substition-permuation network. These networks work by chaining together multiple rounds of substitution and permutation operations, and in each round the (expanded) key is added through a bitwise XOR-operation

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

Figure 2.1: The AES State Matrix

AES has three different versions: AES-128, AES-192 and AES-256. All encrypt blocks of data with a length of 128 bits. In Figure 2.1 above, a 128-bit AES data-block is visualized through bytes b_0, b_1, ... , b_{15}. Note that, rather counterintuitively, the state matrix is filled column-by-column, top-to bottom, instead of row-by-row, left-to-right. The difference between the versions is in the key length: 128 bits for AES-128, 192 for AES-192 and 256 for AES-256. Another difference is the amount of encryption rounds. AES-128 has 10 rounds of encryption, AES-192 has 12 rounds, and AES-256 14. Each of these rounds consists of four consecutive operations: SubBytes, ShiftRows, MixColumns and AddRoundKey. Before starting the first round of encryption, the key is added to the plaintext using a bitwise XOR operation. For all key lengths, the MixColumns operation is omitted in the last round. In the sections below, SubBytes, ShiftRows, MixColumns and AddRoundKey are described in

more detail. A block diagram of AES-128 can be seen in Figure 2.2. From now on, when referring to AES, AES-128 is intended.

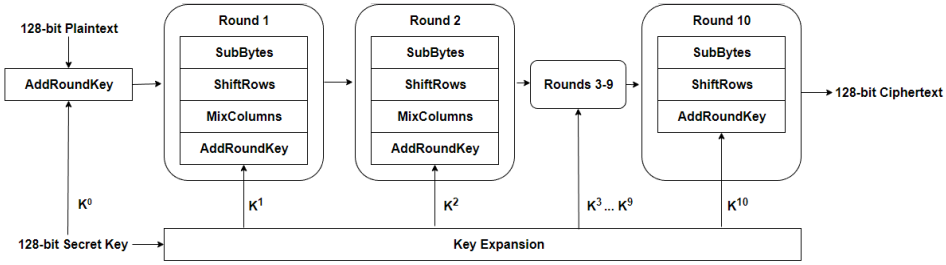


Figure 2.2: A Block Diagram of AES-128

2.2 SubBytes

SubBytes is responsible for providing confusion in the AES algorithm. It works by mapping each the 16 bytes of the 128-bit input to another, fixed byte. This so-called S-box is designed to be as non-linear as possible, to ensure maximal resistance against cryptanalysis. Furthermore, the designers of Rijndael argue that the rest of the algorithm is strong enough to provide secure encryption, even when the S-box is replaced by a less ideal (more linear) one. In Figure 2.3 below, a visual representation of SubBytes is presented, and in Figure 2.4 the contents of the S-box can be seen. When, for example, the input of SubBytes is 0x 00112233 44556677 8899AABB CCDDEEFF, the output becomes 0x 638293C3 1BFC33F5 C4EEACEA 4BC12816. The confusion property of the S-box is can be clearly observed here.

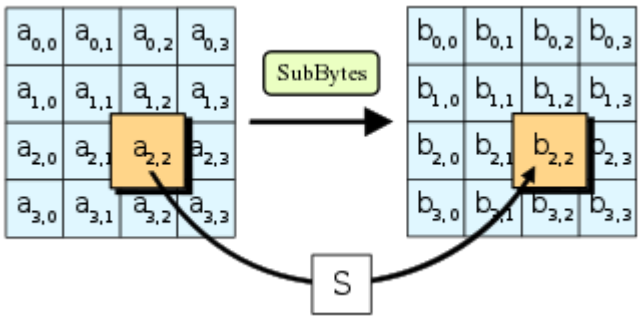


Figure 2.3: AES SubBytes [3]

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 2.4: The Contents of the AES S-box

2.3 ShiftRows

ShiftRows is the first of two steps in AES providing diffusion. It works by performing a cyclical left-shift on each of the rows of the state matrix. Each row gets shifted n times to the left, with n being the row index. ShiftRows ensures that MixColumns doesn't operate on each column independently, which would effectively render AES a 32-bit encryption algorithm, thus providing greater diffusion. Figure 2.5 visually depics the ShiftRows operation. Using the same example as in Chapter 2.2: 0x00112233 44556677 8899AABB CCDDEEFF at the input of ShiftRows yields 0x0055AAFF 4499EE33 88DD2277 CC1166BB.

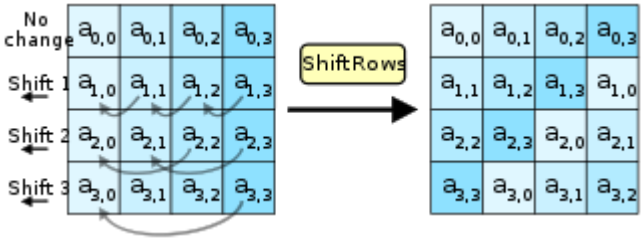


Figure 2.5: AES ShiftRows [3]

2.4 MixColumns

MixColumns is the second step in providing diffusion. After ShiftRows, MixColumns performs a linear transformation on the columns of the state matrix by performing a matrix multiplication on them. However, this matrix multiplication is different from typical matrix multiplication, as the multiplication is done in the Galois Field, GF(2^8). This means the additions become XOR operations, and multiplications become a complex operation, with the column being treated as a 7th grade polynomial. In Figure 2.6 a visualisation of MixColumns can be seen. As an input-output example, consider the output of the example used in Chapter 2.3: 0x0055AAFF 4499EE33 88DD2277 CC1166BB, which yields 0xAAB0001A E5774FDD 22388892 6DFFC755. Through ShiftRows and MixColumns, the information in the exemplatory input is neatly diffused.

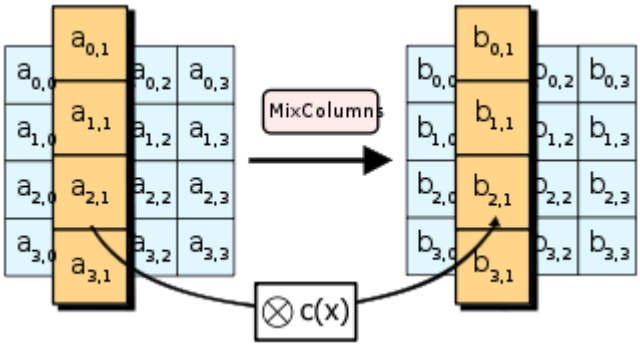


Figure 2.6: AES MixColumns [3]

2.5 AddRoundKey

In the last step of the round, the secret key is added. AES uses a unique key for each round, which is derived from the main 128-bit key through the AES keyschedule. In Equation 2.1 below, a mathematical representation for the computation of the round keys is shown. K_0, K_1, K_2, K_3 denote the 4 32-bit words of the original 128-bit key. W_0, W_1, ... , W_{43} represent the 44 words of the expanded key. rc_i represents a round constant. The values for rc_i can be found in Table 2.1. SubWord is the AES S-box applied to each of the 4 bytes of a word; RotWord a one-byte left cyclical shift, similar to ShiftRows.

$$W_i = \begin{cases} K_i & \text{if i} < 4 \\ W_{i-4} \oplus SubWord(RotWord(W_{i-1})) \oplus rc_{i/4} & \text{if i} \geq 4 \text{ and i mod } 4 = 0 \\ W_{i-4} \oplus Wi - 1 & otherwise \end{cases} \tag{2.1}$$

i	1	2	3	4	5	6	7	8	9	10
rc_i	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80	0x1B	0x36

Table 2.1: Round constants for the AES-128 key schedule

The round keys derived using the keyschedule are added to the output of MixColumns by a bitwise XOR operation, making decryption impossible without the key. The output of AddRoundKey is used as the input of the next round, due to the cyclical nature of the algorithm. The algorithm can then be repeated the desired number of rounds. AddRoundKey can be seen in Figure 2.7.

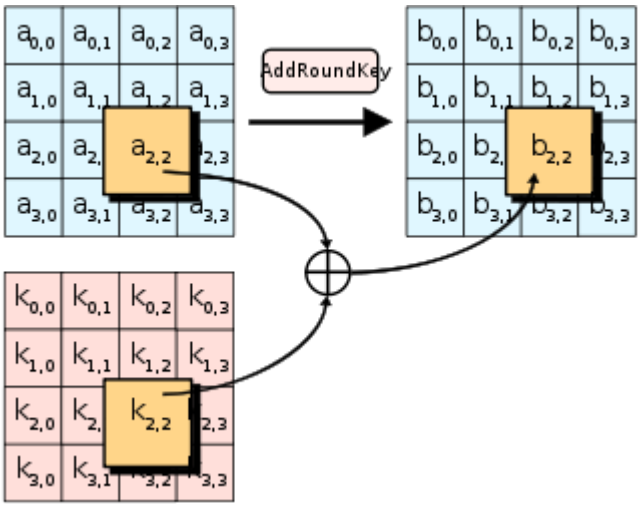


Figure 2.7: AES AddRoundKey [3]

3 Differential Fault Analysis (DFA)

Differential Fault Analysis (DFA) is a technique used to reveal information about the internals of a black-box cryptographic implementation. An attacker induces a fault somewhere in the cryptosystem, and observes the output. This is repeated multiple times, until enough faulty ciphertexts (and the correct one) are collected. The attacker then uses publicly available knowledge of the algorithm to determine the secret key of the cryptosystem. Biham and Shamir [4] laid the foundation for this in their 1997 paper. Based on their work, numerous attacks based on DFA have been published. In this chapter, two examples of DFA are presented: the Giraud attack, and a more generalized model. These two examples will later be used to demonstrate DAVOS' ability to emulate these attacks on an FPGA-based implementation of AES. Two step-by-step plans for an attacker to perform a FIA and DFA are presented, along with several terminal programs to automate these steps.

3.1 The Giraud Attack

In 2004 Giraud published his paper 'DFA on AES' [5]. In this work he presents a method of performing DFA by targeting the 9th round of AES. When flipping a bit between the 9^{th} round MixColumns and the 10^{th} round SubBytes, the output will show a one-byte mismatch with the correct cipher. In Figure 3.1 the propagation of this bit flip is visualised. Giraud proved that in 97 % of cases, three unique bits in each byte of the intermediate ciphertext must be flipped to uniquely identify the corresponding byte of the tenth round key. Thus, a Giraud attack can uniquely identify the entire secret AES key using 48 faulty ciphertexts, i.e. 3 faulty ciphers with a one-byte mismatch in each of the sixteen bytes of the ciphertext. Since it is known that at the input of the 10^{th} round SubBytes a one-bit mismatch is present, the amount of possible values for the correct (non-faulty) input of the 10^{th} round SubBytes is reduced significantly, allowing the attacker to retrieve the secret key via brute-force.

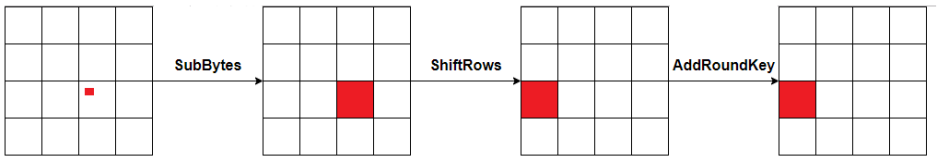


Figure 3.1: Propagation of a single-bit fault induced at the input of the 10^{th} round of AES

Giraud described a correct execution of the 10^{th} round with Equation 3.1, and a faulty execution with Equation 3.2:

$$C_{ShiftRow(i)} = SubBytes(M_i) \oplus K^{10}_{ShiftRow(i)} \tag{3.1}$$

$$D_{ShiftRow(i)} = SubBytes(M_i \oplus e_i) \oplus K^{10}_{ShiftRow(i)} \tag{3.2}$$

All subscripts are byte indices. Furthermore, i denotes the byte index at the start of the 10th encryption round, M_i is the correct intermediate ciphertext at the start of the 10^{th} round, K^{10} the tenth round key and e_i is the induced fault at the start of the tenth round. Since the Giraud attack flips one bit at the start of the tenth round, e_i must have a Hamming weight of 1, i.e. e_i = 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80. Thus, an attacker attempting to find a byte K^{10}_i of the 10^{th} round key can follow the steps below:

- Find the correct ciphertext C
- Find three unique faulty ciphertexts with a one-byte fault in the same byte D^1, D^2, D^3
- For all 256 candidates for K^{10}_i, perform the inverse of the final round on Ci to find the potential correct inputs of the 10th round $M_{iShiftRow(i)}$ using Equation 3.3: (iShiftRow denotes inverse ShiftRow, iSubBytes inverse SubBytes)
$$M_{iShiftRow(i)} = iSubBytes(C_i \oplus K^{10}_i) \tag{3.3}$$
- For all 256 candidates for K^{10}_i, perform the inverse of the final round on D^1_i to find potential faulty inputs of the 10th round, which must satisfy Equation 3.4:
$$M_{iShiftRow(i)} \oplus e_{iShiftRow(i)} = iSubBytes(D_i \oplus K^{10}_i) \tag{3.4}$$
- Reduce the set of candidates for K^{10}_i by only keeping those candidates for K^{10}_i which satisfy Equation 3.4 above.
- Repeat the previous two steps for D^2 and D^3 with the remaining candidates for K^{10}_i. As proven by Giraud, there is a significant chance only one candidate remains. If this is not the case, another faulty ciphertext D^4 can be used.

  A terminal program that automates the steps described above can be found in Appendix A.1.

  An attacker can repeat these steps for all 16 bytes of the key K^{10}_i, after which the inverted key schedule can be applied to obtain the key K, using the code in Appendix A.3. All of the above steps will be demonstrated in Chapter 6.5.

3.2 Generalized attacks on AES

Differential Fault Analysis on AES can also be reduced into a more general format. Moradi, Shalmani and Salmasizadeh have done just that. In their paper 'A Generalized Method of Differential Fault Attack Against AES Cryptosystems' [6], they present a model for deriving the secret key from an AES implementation, by analyzing the possible ways a fault manifests itself at the input of the 10^{th} round SubBytes. With the fault model, the correct ciphertext and a sufficient amount of faulty ciphertexts the tenth round key K_i^{10} can then be derived through elimination.

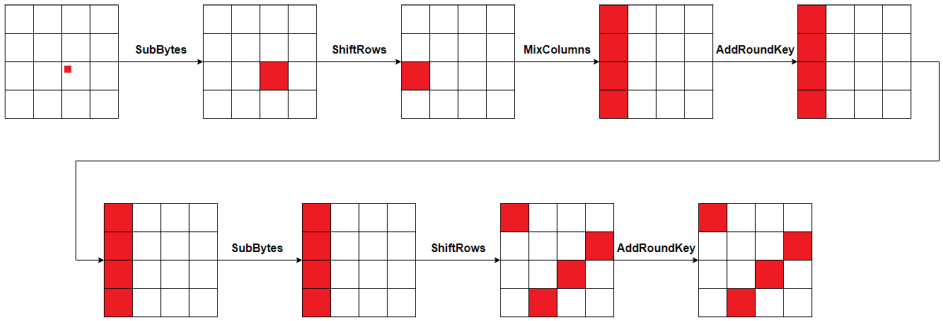


Figure 3.2: Propagation of a single-bit fault induced at the input of the ninth round of AES

This model can be used in, for example, an attack occuring between the output of the 8th round MixColumns and the input of the 9^{th} round MixColumns. In Figure 3.2 it can be seen that such a fault would manifest itself in the ciphertext as a 4-byte mismatch. By analyzing the propagation of such a fault, an attacker is able to reduce the search space for key candidates significantly. For such an attack, the model in Equation 3.5 below can be used.

$$\text{SubBytes}\left(\begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \end{bmatrix}\right) \oplus SubBytes\left(\begin{bmatrix} I_1 \oplus e_1^{'} \\ I_2 \oplus e_2^{'} \\ I_3 \oplus e_3^{'} \\ I_4 \oplus e_4^{'} \end{bmatrix}\right) = \begin{bmatrix} e_1^{"} \\ e_2^{"} \\ e_3^{"} \\ e_4^{"} \end{bmatrix} \tag{3.5}$$

Here, the set I = I_1 & I_2 & I_3 & I_4 contains all possible candidates for the input of the 10^{th} round SubBytes. I is a word; I_1 through I_4 are bytes. The same applies to e' and e". e' contains all possible ways a fault covered by the fault model can manifest itself at the input of the 10^{th} round SubBytes. e" is the fault observed at the output of SubBytes, and thus also at the output,

after ShiftRows is accounted for. By gathering faulty ciphertexts showing 4-byte mismatches, the attacker can use the model previously presented to reduce the amount of entries in the set I. To accomplish this, the full range of possible effects of an injected fault must be identified in the set e', and enough faulty ciphertexts must be gathered to reduce the set I to only one candidate.

For an attack that flips a bit between the output of the 8^{th} round MixColumns and the input of the 9^{th} round MixColumns, the 4 bytes that show a mismatch at the output are always the same, as the 10^{th} round ShiftRows always affects the propagation of the fault in the same way. This can easily be seen in Figure 3.2. In Table 3.1 the affected output bytes for the targeted column of the AES state matrix are shown.

Affected column at 9^{th} round MixColumns input	Affected bytes before 10^{th} round ShiftRows	Affected bytes after 10^{th} round ShiftRows
1	0, 1, 2, 3	0, 13, 10, 7
2	4, 5, 6, 7	4, 1, 14, 11
3	8, 9, 10, 11	8, 5, 2, 15
4	12, 13, 14, 15	12, 9, 6, 3

Table 3.1: Propagation of affected bytes for a fault in the 9^{th} round MixColumns input

To determine all elements of the set e', AES can simply be applied to all possible faults. Since the targeted column can be easily determined, this is reduced to 32 possibilities, assuming a single bit is flipped. Assuming the output of 8^{th} round MixColumns is targeted (faults injected after 9^{th} round SubBytes are a subset of this model), the output of 9^{th} round SubBytes shows a 1-byte mismatch. Due to the non-linearity of SubBytes, its value is unknown, however, only one byte is faulty. ShiftRows can be ignored, as in Table 3.1 it is shown that the affected column is easily determined from the faulty output. AddRoundKey is ignored due to the fact that faults propagate through an XOR operation. Thus, all faults under this model are covered by Equation 3.6:

$$e^{'} = MixColumns(e) \tag{3.6}$$

The set e contains all possible errors in the 9th Round SubBytes; i.e. e = 0x00000001, 0x00000002, 0x00000003, ... , 0x000000FF, 0x00000100, ... , 0x0000FF00, ..., 0xFF000000.

In order to find four bytes of K^{10} using this fault model, an attacker would carry out the following steps:

- Find the correct ciphertext C
- Gather enough faulty ciphertexts D with 4-byte mismatches to uniquely determine the input of the 10th Round SubBytes
- Find $e^{"} = C \oplus D$.

- Byte-by-byte, check all 2^{32} candidates in I. Discard entries that do not satisfy the model.
- Repeat the previous two steps with the remaining candidates in I until only one candidate is left.
- Determine $K^{10_0,13,10,7} = I \oplus C_{0,13,10,7}$. Use Table 3.1 for correct indices for other bytes, or perform inverse ShiftRows.

A terminal program that automates the first five steps above can be found in Appendix A.2. The attacker can repeat these steps for columns 2, 3, and 4. Once all bytes of K^{10} are known, the inverse key schedule can be applied to extract the original key. This strategy will be used in Chapter 6.6.

Given enough computational prowess, this strategy can be expanded to an even more inaccurate attack. In that case the attacker knows nothing about e': 2^{128} entries. Moradi, Shalmani and Salmasizadeh have demonstrated in [6] that it is possible to obtain the key using these fault models, thus allowing the attacker to obtain the secret key of a cryptosystem with an extremely inaccurate attack method.

4 Fault Injection Attacks (FIAs) on FPGAs

4.1 FPGA Architecture

In order to effectively carry out a FIA through DFA on an FPGA-based implementation of a cryptographic algorithm, intricate knowledge of the inner workings of an FPGA is required. FPGAs are made up of a variety of Programmable Logic Blocks, e.g. Configurable Logic Blocks (CLBs), DSP Blocks and RAM blocks. FPGAs have thousands of these; the xc7z020 has around 85.000. [7] Every one of these cells contains multiple Basic Elements of Logic (BELs), which makes them capable of performing different kinds of logic functions. Examples of BELs include Flip-Flops, Look-Up Tables (LUTs), RAM blocks or DSP blocks.

BELs are arranged in groups of related blocks, known as slices. Slices are identifyable by their coordinates on the device. A group of related slices forms a tile, which is capable of implementing one of the FPGA's Programmable Logic Block functions. Tiles are arranged in columns, forming a Clock Row. A 2D array of tiles driven from the same clock source is called a Clock Region.

The FPGA's logic can be connected through interconnects, allowing the user to program the FPGA with a hardware description, often written in a HDL such as VHDL or Verilog. After the user generates a bitstream to program their target device with in a design suite, e.g. Vivado, this bitstream can be loaded into the FPGA's configuration memory, to program the device with.

4.2 Practical Feasibility

In order to carry out an FIA in practice, an adversary would most certainly require physical access to the device. Once physical access is secured, the attack can be performed. FIAs can be performed in multiple ways, each bringing their own balance between precision and complexity. For example, an attacker could use a highly precise laser to flip a bit at the desired location and time [8], allowing him to carry out the Giraud attack described in Chapter 3.2. However, such a laser attack setup is highly complex and expensive, meaning only a technically advanced adversary could carry out such an attack. Examples of simpler ways of tampering with the internals of an FPGA-based encryption algorithm are clock and power glitches [9]. These attacks are less complex to perform, yet lack the precision of a laser-based method. The attacker would thus have to use a more general fault model, as described in Chapter 3.3. Another way of performing FIAs is through the FPGA's configuration file, the bitstream. An attacker with inside knowledge of the design could reverse-engineer the bitstream and re-program the device with a compromised bitstream, allowing him to retrieve information about the key.

5 Countermeasures against FIAs

The increased attention for attacking FPGA-based designs has not gone unnoticed. The industry itself has taken measures. Xilinx' 7-series FPGAs, for example, make use of bitstream encryption and authentication when programming the device. [10] However, as is often the case, this is a cat-and-mouse game. For example, Yi [11] has shown how to circumvent the encryption of the bitstream. Aside from the industry, developers can also take action themselves, by making their designs more robust against aforementioned attacks. Some possibilities that designers have are discussed in this chapter. Furthermore, a countermeasure is implemented, which can be used to demonstrate DAVOS' use in emulation of FIA attacks.

5.1 Redundancy

The most trivial solution to protect FPGA-based cryptographic implementations from (single) bit-flip FIAs is to use redundancy. Redundancy can be done either in area or time, depending on the design constraints. In all redundancy-based protection measures, the encryption algorithm is executed multiple times, either in parallel (area redundancy) or in series (time redundancy). The results from multiple executions of the algorithm are passed on to a voter, which decides the output. When it comes to FIA protection using redundancy, the voter could also set an error flag, to alert the owner the device is potentially being tampered with. While a countermeasure that makes use of redundancy provides 100% fault coverage when targeting a single element of the design, it is obvious that the main drawback of redundancy is the increased resource overhead, either in time or in area. These resources may not always be available, due to constraints set by the required output bitrate or limited CLBs on the FPGA. An example of a FIA countermeasure based on redundancy in time is the Built-in Self-Test by Redonet Klip [11]. In this research an area-based Triple Mode Redundancy countermeasure is presented, which can be used as an example of DAVOS's ability to emulate FIAs against a user-developed countermeasure. In Figure 5.1 below, the designed TMR implementation is presented as a block diagram. The design of the voter is in Figure 5.2. The AES Core is a round-based implementation of AES-128. VHDL code for the top-level TMR entity and the voter are in Appendices B.1 and B.2, respectively.

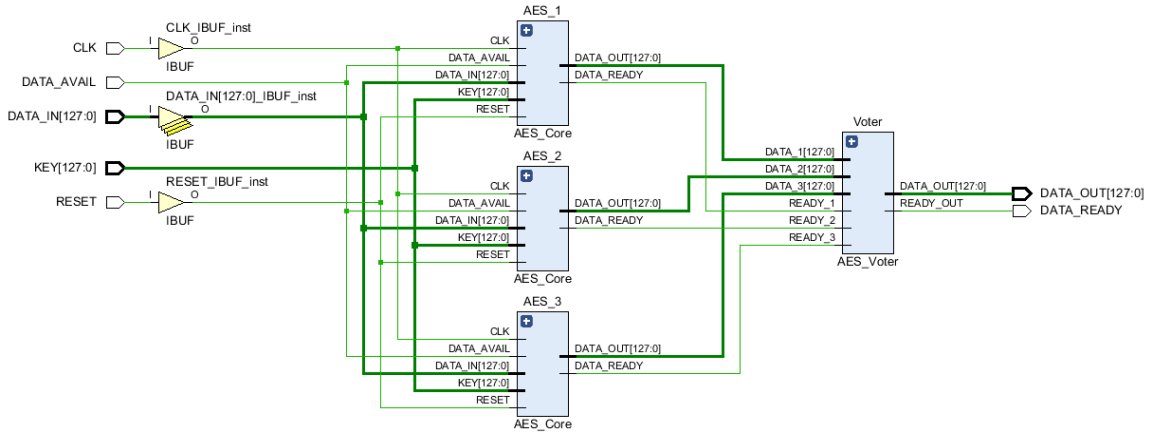


Figure 5.1: Vivado elaborated design of top-level entity AES_TMR

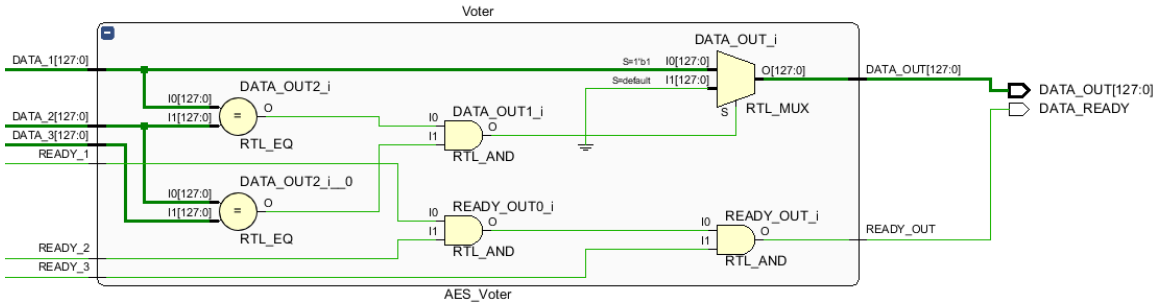


Figure 5.2: Vivado elaborated design of entity AES_Voter

The developed voter differs slightly from a typical TMR system. Usually the voter is a majority voter. The developed voter switches the output to zero if the three outputs of the AES instances do not match perfectly. Thus, the output is switched to zero if a fault is injected in any location in any of the three AES cores. This allows a successful to be easily spotted; the output is all-zeroes, instead of the correct cipher if a majority voter had been used. The control signal DATA_READY is generated asynchronously from the three READY outputs of the AES cores. They are passed through a triple and-gate: only if all three signals are high, the output DATA_READY is set high.

5.2 Advanced Solutions

More advanced solutions to protect cryptographic systems from Fault Injection attacks also exist. These typically make use of concurrent error detection (CED), to detect faulty operation of the algorithm, and alerting the owner. A notable example of such a solution is ParTI [13], an excellent countermeasure achieving fault coverage of over 90% at a mere 12% increased area overhead.

Another interesting solution that is not further explored in this research, but is worth mentioning, is replacing the S-box. As mentioned in Chapter 2, the authors of Rijndael argue that AES is strong enough to provide resilient encryption, even when the S-box is less than ideal. This means one could replace the S-box with a custom one. The combination of an unknown S-box and an unknown secret key would create a greater challenge for an adversary to crack the system with DFA. The methods described in Chapter 3 would be rendered useless, as these assume the S-box to be known and constant.

6 Bitstream-based FIA emulation using DAVOS

This work aims to develop a platform used in aiding FPGA developers test their designs against FIAs. The reliability assessment kit DAVOS [14] has been modified to work as an emulator for FIAs through the bitstream, re-programming the device as if it were affected by a successful FIA.

6.1 Bitstream manipulation and reverse engineering

In order to effectively emulate FIAs, the bitstream of the target device must first be thoroughly understood. With an in-depth understanding of the bitstream, the user can target and invert single bits in their design. If the modified bitstream is then used to program the device, a single-bit FIA is effectively emulated. This process can then be repeated as many times as desired, to perform a FIA or to test the robustness of a design against FIAs. In this research, the ZedBoard [15] is used, which has an xc7z020 device onboard. The bitstream of the 7-series devices has been thoroughly explored in [16].

6.2 Existing FIA emulation

Research has been done into bitstream fault injection (BiFi), as well as into automating the testing of robustness of cryptographic implementations. Yi [11] has demonstrated a succesful BiFi, though Yi's work only targets LUTs. However, this does show that emulation of FIAs through the bitstream is possible. Recently a tool for testing the robustness of cryptographic implementations has been published: VerFi [17]. VerFi aims to automatically evaluate the robustness of cryptographic implementations, giving developers insight into the behavior of their design in the face of FIA attacks. VerFi determines the Fault Coverage Rates for countermeasures against FIA, giving developers quick insight into how effective their countermeasures are. VerFI, however, only has simulation-level functionality; no physical target device is programmed.

6.3 DAVOS as an FIA emulation tool

The works mentioned in the previous section could be combined into one powerful tool. This is where DAVOS comes in: Like Yi's work, it has the ability to upload modified bitstreams to a device, effectively emulating FIAs. Furthermore, it is not only limited to LUTs. Therefore, combining VerFi's automated robustness verification with DAVOS' hardware-based FIA emulation possibilities could allow users to obtain fault coverage rates for their design, running on the target device. DAVOS is an EDA toolkit aimed at reliability assessment. Similarly to FIAs, the injection of faults is an important aspect of reliability analysis. DAVOS aims to make reliability analysis easier by providing functionality that allows designers to target any aspect of their design, at any moment, both in simulation and in hardware. DAVOS does this by re-programming the device

using a modified bitstream during runtime. This capability makes DAVOS interesting for emulating FIAs on FPGA-based cryptographic implementations as well. DAVOS can perform FIAs in both simulation, using software like QuestaSim, and hardware, by physically programming an FPGA like the xc7z020. In this research the emphasis is on DAVOS' ability to emulate FIAs in hardware, as this is a novelty. DAVOS' simulation capabilities can prove useful as well, though, for example when attempting to find out the specific injection time for the desired fault. DAVOS has the ability to automatically perform the bitstream manipulation described in Chapter 6.1. It can do so for a large number of targets. The modified bitstreams are uploaded sequentially to the device, and the output is observed. Through its configuration file the user can specify the design to be targeted, the type of logic to be targeted, the timing of the injection and the amount of injections to be performed. Through its configuration file the user can specify in detail what aspects of the design should be targeted. For FIAs, there are three important aspects that DAVOS can control. Firstly, the user can specify the type of logic to be targeted, i.e. flip flops, LUTs, BRAM, etc. Secondly, as DAVOS is able to pause the clock and re-program the device at any moment during execution, the user can control exactly when the fault is injected. Finally, the user can specify the exact scope within the design to be targeted, allowing for a wide range of scenarios to be executed. For example, the user can emulate an attack on 'the flip flops at the input of the 10th round SubBytes', i.e. a Giraud attack. All the user has to know is the timing ('When does the 10th round start?'), and the scope of the input of SubBytes. DAVOS observes the output using a finished-flag, which is set high whenever the output should be read. The user can then analyse the output provided by the Report Builder, and check if any faulty ciphertexts have propagated to the output, which would allow an adversary to perform DFA and crack the key of the cryptosystem. Having all aforementioned capabilities, DAVOS shows potential as an FIA emulator.

Critical in emulating FIA attacks with DAVOS are the configuration parameters. The target scope and type of logic can be easily determined using Vivado's elaborated design. However, the injection time proves to be more complicated. ZedBoard operates using an AXI interconnect to connect the processing system (PS), in this case DAVOS, and the programmable logic (PL), in this case the AES instance performing the encryption. The behaviour regarding timing of this system is unknown. However, DAVOS has the ability to stop the system clock, allowing the control signal DATA_AVAIL to be set high asynchronously, after which the clock is resumed. This allows DAVOS to reconfigure the PL after a user-specified time period after setting DATA_AVAIL high. After this, the required delay for executing e.g. a Giraud attack can be experimentally determined.

6.4 Methods

In the next sections a proof-of-concept demonstration of DAVOS' usability as a FIA emulation tool is presented. The strategy used to emulate the FIA attacks on the xc7z020 was as follows:

- Determine the required parameters to emulate the desired attack (target logic, injection time, target scope)
- Configure DAVOS via the configuration file
- Run DAVOS and build the report
- Analyse the output data
- If enough faults have propagated to the output, derive the secret key using DFA on the gathered faulty ciphertext

6.5 Emulating a Giraud attack

DAVOS was used to target the flip flops at the input of the 10th round in a round-based AES implementation. Enough faulty ciphertexts were gathered to successfully perform a Giraud attack, i.e. three faulty ciphertexts per byte, totalling 48. The code in Appendix A.1 was used to derive the corresponding key bytes K10. Once the full tenth round key was derived, the inverse key schedule was applied to derive the original key. In Table 6.1 the full result can be seen. The notations are the same as in Chapter 3.2. C is the correct ciphertext. For compactness, the 48 faulty ciphertexts have been compressed into three rows, D^1, D^1 and D^1. K^{10} is the tenth round key. Key is the secret key, which can be found by applying the inverse key schedule to K^{10}. The terminal program used to find the key from K^{10} is in Appendix A.3.

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	3A	D7	8E	72	6C	1E	C0	2B	7E	BF	E9	2B	23	D9	EC	34
D^1	90	7D	71	AD	54	2D	82	BD	B4	03	1E	63	A0	47	A0	FE
D^2	F6	1C	5D	D4	17	75	BD	35	B6	21	2F	EB	90	9D	74	6F
D^3	2C	EA	93	C4	05	C3	72	89	F4	93	A5	-	8C	E3	35	43
K^{10}	B4	EF	5B	CB	3E	92	E2	11	23	E9	51	CF	6F	8F	18	8E
Key	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Table 6.1: An emulated Giraud attack.

In Table 6.1 it can be seen that the key used in the experiment was all-zeroes. Note that only two faulty ciphertexts were used to determine the 11^{th} byte of K^{10}. Due to the random nature of DAVOS' fault injections, only two faulty ciphertexts were obtained for this byte. However, these were enough to uniquely determine the 11^{th} byte of K^{10}. By decrypting the ciphertext C with the key of all-zeroes, the input plaintext can also be found: 0x 80000000 00000000 00000000 00000000.

6.6 Attacking the input of the 9^{th} round

DAVOS was used to gather the required faulty ciphertexts as described in Chapter 3.3, to perform an attack on the input of the 9^{th} round of the same AES implementation used in Chapter 6.5, using the same settings as in the previous section, but injecting the faults one round earlier. The code in Appendix A.2 was used to reduce the amount of key candidates, and it turned out that two four-byte mismatching ciphers were enough to result in a single unique candidate for the relevant four bytes of the tenth round key K^{10}. Thus, using 8 faulty ciphertexts the key could be found. In Table 6.2 the result of the emulated attack is presented. I represents the input of the K^{10} round SubBytes. The 8 faulty ciphertexts have been reduced into D^1 and D^2 for compactness. Related bytes (Table 3.1) are indicated by using boldface (fault occured in column 1), italic (fault in column 2), underlined (column 3) and normal text (column 4).

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	3A	D7	8E	72	6C	1E	C0	2B	7E	BF	E9	2B	23	D9	EC	34
D^1	**10**	*C7*	EA	51	*E3*	4A	5E	**66**	4E	33	**D1**	*DA*	2F	**17**	*12*	A8
D^2	**D6**	*C4*	85	71	*7A*	A8	91	**FF**	61	9A	**63**	*6B*	3E	**05**	*19*	90
I	**E6**	**B9**	**9A**	**A2**	*48*	*76*	*BA*	*AE*	8D	F0	B5	C0	5D	B9	94	DB
K^{10}	B4	EF	5B	CB	3E	92	E2	11	23	E9	51	CF	6F	8F	18	8E
Key	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Table 6.2: An emulated attack on the input of round 9

After determining the input of the 10^{th} round SubBytes (in row I), K^{10} was determined using the theory in Chapter 3.3, after which the inverse key schedule was applied. Because the input cipher and the key were not modified in the software, the key and the correct ciphertext are the same as in the previous section, as expected.

6.7 Attacking a TMR implementation with DAVOS

Attempts to attack the TMR implementation in Figure 5.1 using DAVOS. However, at the moment of writing this thesis, this has not yet been done succesfully. Problems arose with the packaged vivado IP and the control signals DATA_AVAIL and DATA_READY. DAVOS uses DATA_READY as the finished-flag to read the output data. It is suspected that this flag is not being set properly, causing the output not to be logged.

7 Evaluation

In the previous section DAVOS has been used as an FIA emulator. As intended, DAVOS can insert both transient and permanent faults in any location of the design. It should be noted that, for emulation of permanent faults, adequate modifcation of the DATA_READY signal should be done. Currently, difficulties arise when attempting to emulate faults in a design different from the default design. Faults can be emulated as intended on the default design (Chapter 6.4 and 6.5), but difficulties arise when testing a different design. Fixing this is key to using DAVOS as an FIA emulator in practice.

Furthermore, if the FIA emulation on a TMR implementation had succeeded, no faults would likely have propagated to the output, as TMR provides a 100% Fault Coverage Rate when only one bit is flipped, at the cost of 200% area overhead. However, if the functionality of DAVOS is expanded so that multiple bits in a design can be targeted, TMR can be compromised. Assuming a technically advanced adversary with access to multiple highly accurate lasers and intricate knowledge of the TMR design in place, such a scenario would be plausible.

	Yi	VerFi	DAVOS
Simulation FIA	No	Yes	Yes
Hardware FIA	LUT-only	No	Yes
Fault Coverage Analysis	No	Yes	No

Table 7.1: Comparison between Yi's FIA, VerFi and DAVOS

When comparing DAVOS to previous work on bitstream-based FIA attacks and FIA emulators, DAVOS shows great potential. In Table 7.1 a comparison is made between the work of Yi[11], VerFi[17] and DAVOS. With the right modifications, DAVOS can combine VerFi's analysis capabilities with the ability to do emulations in hardware, creating a complete FIA emulation environment for developers looking to test the robustness of their cryptographic implementations.

8 Conclusions and Recommendations

In this thesis DAVOS was used to emulate highly precise FIAs on FPGA-based cryptographic implementations. DAVOS' ability to perform single bit-flip attacks on user-designed cryptographic is shown. Since DAVOS can be used to target any bit in any design at any time, it is a suitable tool for developers of cryptosystems to test their system in the face of an actual FIA. DAVOS was used to extract the key from a simple round-based AES implementation using two different target scopes. Both of these attacks were executed succesfully, using fault models derived in Chapter 3. This shows that DAVOS is indeed usable as an FIA emulation tool. Furthermore, an attempt was made to attack an implementation of AES was attacked with an ideal countermeasure was in place. Despite not succeeding in the latter attack, the other results provide a good proof-of-concept of DAVOS' usability in FIA emulation, escaping the simulation-only environment of current state of the art tooling such as VerFi.

DAVOS shows great potential to be used to emulate FIAs. Currently, the main drawbacks are in usability. This could be expected, as DAVOS is not used for its main purpose. In this research the analysis of the output data was done by hand from the output report provided by DAVOS, with the assistance of terminal programs. Furthermore, the user-friendliness of the environment is limited, as proven by the ongoing struggles to attack TMR. Also, only a single bit can be flipped at a time. Thus, future research could focus on improving this. For this, VerFi [17] could be used as inspiration. If the capabilities of DAVOS are expanded in such a way that any cryptographic implementation can be easily attacked in multiple locations, and Fault Coverage Rates for FIA countermeasures are provided, a potent FIA emulation environment would be created.

9 Bibliography

[1] J. Daemen and V. Rijmen, "AES proposal: Rijndael", September 1999

[2] C. Shannon, "Communication Theory of Secrecy Systems", Bell System Technical Journal, vol. 28(4), pp. 708-710, 1949

[3] Wikipedia, "Advanced Encryption Standard" [Online], Available: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[4] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," CRYPTO '97 Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology, pp. 513–525, October 1996

[5] C. Giraud, "DFA on AES", International Conference on Advanced Encryption Standard, pp. 27-41, 2004.

[6] A. Moradi, M.T. Manzuri Shalmani and M. Salmasizadeh, "A Generalized Method of Differential Fault Attack Against AES Cryptosystem", CHES 2006, pp 91-100, 2006

[7] Xilinx, "Zynq-7000 SoC Data Sheet: Overview" [Online], Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[8] S.P. Skorobogatov and R.J. Anderson, "Optical Fault Induction Attacks", CHES 2002, pp. 2-12, 2002

[9] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice and countermeasures", Proceedings of IEEE, pp. 3056–3076, 2012.

[10] K. Wilkinson, Xilinx: "Using encryption to secure a 7 series FPGA Bitstream" [Online], 2018, Available: https://www.xilinx.com/support/documentation/application_notes/xapp1239-fpga-bitstream-encryption.pdf

[11] Y. Ji, "Protection of FPGA-based Cryptographic Implementations against Fault Injection Attacks", MSc Thesis, Friedrich-Alexander Universität Erlangen-Nü, December 2017

[12] T.H. Redonet Klip, 'A Built-In Self-Test as a Countermeasure for Fault Injection Attacks on Cryptographic Devices', BSc Thesis, University of Twente, July 2018

[13] T. Schneider, A. Moradi and T. Güneysu, "ParTI – Towards Combined Hardware Countermeasures Against Side-Channel and Fault-Injection Attacks", CRYPTO 2016, pp. 302-332, 2016

[14] I. Tuzov, D. de Andres and J. Ruiz "DAVOS: EDA Toolkit for Dependability Assessment, Verification, Optimisation and Selection of Hardware Models", 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2018

[15] AVnet, "Zynq Evaluation and Development Hardware User's Guide" [Online], Available: http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

[16] K.D. Pham, E. Horta and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations", DATE 2017, pp. 894-897, 2017

[17] V. Arribas, F. Wegener, A. Moradi and S. Nikova, "Cryptographic Fault Diagnosis using VerFI", Cryptology ePrint Archive, Report 2019/1312, 2019

A Key extraction and expansion

In this appendix the C code used for extracting the secret key in Chapter 6 is presented. It's not the most elegant solution, but it gets the job done.

A.1 Byte-wise key extraction used in the Giraud attack

The C code below extracts one candidate byte of the key when faulty ciphertexts matching the fault model for the Giraud attack is obtained. Compile it, run it, then enter C, D^1, D^2 and D^3, and the relevant byte of K^{10} is displayed on the terminal. For example: Entering 3A, 90, F6, 2C will yield B4: byte 1 of Table 6.1.

```
#include <stdio.h>
#include <stdint.h>

//The inverse AES S-box
static const uint8_t invsbox[256] = {
  0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb,
  0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb,
  0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e,
  0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25,
  0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92,
  0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84,
  0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06,
  0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b,
  0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73,
  0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e,
  0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b,
  0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4,
  0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f,
  0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef,
  0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61,
  0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d };

//Onecount finds the hamming weight of the function argument
  unsigned int onecount(unsigned int n)
{
    unsigned int count = 0;
    while (n) {
        count += n & 1;
        n >>= 1;
    }
    return count;
}
```

```
int main() {

//Variable declaration
   unsigned int C = 0x00, D1 = 0x00, D2 = 0x00, D3 = 0x00;
   unsigned int C9 = 0x00, D91 = 0x00, D92 = 0x00, D93 = 0x00;
   int i = 0;
   int a = 0;

//IO functions
   printf("Enter correct cipherbyte in hex, then ENTER \n");
   scanf("%x", &C);

   printf("Enter three faulty cipherbytes in hex, separated by ENTER. \n");
   scanf("%x", &D1);
   scanf("%x", &D2);
   scanf("%x", &D3);

   printf("Finding key candidates for C = %x, D1 = %x, D2 = %x, D3 = %x: \n", C, D1, D2, D3);

//Find Key Candidates
   for (i = 0; i <256; i++){
        C9 = invsbox[C^i];       /*compute invSubBytes(M10 XOR K10), i.e. M9 for K10 = 0xi*/
        D91 = invsbox[D1^i];
        D92 = invsbox[D2^i];
        D93 = invsbox[D3^i];

        if (onecount(D91^C9) == 1 & onecount(D92^C9) == 1 & onecount(D93^C9) == 1){
            printf("%x\n", i); /*print if all faulty inputs of R10 have distance 1*/
        }

   }

   return 0;
}
```

A.2 Byte-by-byte key extraction used for attacking the 9^{th} round input

The C code below extracts four candidate bytes of the input of the 10^{th} round SubBytes when faulty ciphertexts matching the fault model for the attack on the input of the 9^{th} round are obtained. Compile it, run it, and enter the correct and faulty ciphertexts. For example, entering:
C = 3A-D9-E9-2B
D^1 = 10-17-D1-66
D^2 = D6-05-63-FF
will yield E6-B9-9A-A2, the first four bytes of I in Table 6.2. It should be noted that the order of input MUST be the third column of Table 3.1, or it will NOT work.

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

//AES S-BOX
static const uint8_t sbox[256] = {
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16 };

//GALOIS2 LUT
static const uint8_t galois2[256] = {
    0x00,0x02,0x04,0x06,0x08,0x0a,0x0c,0x0e,0x10,0x12,0x14,0x16,0x18,0x1a,0x1c,0x1e,
    0x20,0x22,0x24,0x26,0x28,0x2a,0x2c,0x2e,0x30,0x32,0x34,0x36,0x38,0x3a,0x3c,0x3e,
    0x40,0x42,0x44,0x46,0x48,0x4a,0x4c,0x4e,0x50,0x52,0x54,0x56,0x58,0x5a,0x5c,0x5e,
    0x60,0x62,0x64,0x66,0x68,0x6a,0x6c,0x6e,0x70,0x72,0x74,0x76,0x78,0x7a,0x7c,0x7e,
    0x80,0x82,0x84,0x86,0x88,0x8a,0x8c,0x8e,0x90,0x92,0x94,0x96,0x98,0x9a,0x9c,0x9e,
```

```
    0xa0,0xa2,0xa4,0xa6,0xa8,0xaa,0xac,0xae,0xb0,0xb2,0xb4,0xb6,0xb8,0xba,0xbc,0xbe,
    0xc0,0xc2,0xc4,0xc6,0xc8,0xca,0xcc,0xce,0xd0,0xd2,0xd4,0xd6,0xd8,0xda,0xdc,0xde,
    0xe0,0xe2,0xe4,0xe6,0xe8,0xea,0xec,0xee,0xf0,0xf2,0xf4,0xf6,0xf8,0xfa,0xfc,0xfe,
    0x1b,0x19,0x1f,0x1d,0x13,0x11,0x17,0x15,0x0b,0x09,0x0f,0x0d,0x03,0x01,0x07,0x05,
    0x3b,0x39,0x3f,0x3d,0x33,0x31,0x37,0x35,0x2b,0x29,0x2f,0x2d,0x23,0x21,0x27,0x25,
    0x5b,0x59,0x5f,0x5d,0x53,0x51,0x57,0x55,0x4b,0x49,0x4f,0x4d,0x43,0x41,0x47,0x45,
    0x7b,0x79,0x7f,0x7d,0x73,0x71,0x77,0x75,0x6b,0x69,0x6f,0x6d,0x63,0x61,0x67,0x65,
    0x9b,0x99,0x9f,0x9d,0x93,0x91,0x97,0x95,0x8b,0x89,0x8f,0x8d,0x83,0x81,0x87,0x85,
    0xbb,0xb9,0xbf,0xbd,0xb3,0xb1,0xb7,0xb5,0xab,0xa9,0xaf,0xad,0xa3,0xa1,0xa7,0xa5,
    0xdb,0xd9,0xdf,0xdd,0xd3,0xd1,0xd7,0xd5,0xcb,0xc9,0xcf,0xcd,0xc3,0xc1,0xc7,0xc5,
    0xfb,0xf9,0xff,0xfd,0xf3,0xf1,0xf7,0xf5,0xeb,0xe9,0xef,0xed,0xe3,0xe1,0xe7,0xe5};

//GALOIS3 LUT
static const uint8_t galois3[256] = {
    0x00,0x03,0x06,0x05,0x0c,0x0f,0x0a,0x09,0x18,0x1b,0x1e,0x1d,0x14,0x17,0x12,0x11,
    0x30,0x33,0x36,0x35,0x3c,0x3f,0x3a,0x39,0x28,0x2b,0x2e,0x2d,0x24,0x27,0x22,0x21,
    0x60,0x63,0x66,0x65,0x6c,0x6f,0x6a,0x69,0x78,0x7b,0x7e,0x7d,0x74,0x77,0x72,0x71,
    0x50,0x53,0x56,0x55,0x5c,0x5f,0x5a,0x59,0x48,0x4b,0x4e,0x4d,0x44,0x47,0x42,0x41,
    0xc0,0xc3,0xc6,0xc5,0xcc,0xcf,0xca,0xc9,0xd8,0xdb,0xde,0xdd,0xd4,0xd7,0xd2,0xd1,
    0xf0,0xf3,0xf6,0xf5,0xfc,0xff,0xfa,0xf9,0xe8,0xeb,0xee,0xed,0xe4,0xe7,0xe2,0xe1,
    0xa0,0xa3,0xa6,0xa5,0xac,0xaf,0xaa,0xa9,0xb8,0xbb,0xbe,0xbd,0xb4,0xb7,0xb2,0xb1,
    0x90,0x93,0x96,0x95,0x9c,0x9f,0x9a,0x99,0x88,0x8b,0x8e,0x8d,0x84,0x87,0x82,0x81,
    0x9b,0x98,0x9d,0x9e,0x97,0x94,0x91,0x92,0x83,0x80,0x85,0x86,0x8f,0x8c,0x89,0x8a,
    0xab,0xa8,0xad,0xae,0xa7,0xa4,0xa1,0xa2,0xb3,0xb0,0xb5,0xb6,0xbf,0xbc,0xb9,0xba,
    0xfb,0xf8,0xfd,0xfe,0xf7,0xf4,0xf1,0xf2,0xe3,0xe0,0xe5,0xe6,0xef,0xec,0xe9,0xea,
    0xcb,0xc8,0xcd,0xce,0xc7,0xc4,0xc1,0xc2,0xd3,0xd0,0xd5,0xd6,0xdf,0xdc,0xd9,0xda,
    0x5b,0x58,0x5d,0x5e,0x57,0x54,0x51,0x52,0x43,0x40,0x45,0x46,0x4f,0x4c,0x49,0x4a,
    0x6b,0x68,0x6d,0x6e,0x67,0x64,0x61,0x62,0x73,0x70,0x75,0x76,0x7f,0x7c,0x79,0x7a,
    0x3b,0x38,0x3d,0x3e,0x37,0x34,0x31,0x32,0x23,0x20,0x25,0x26,0x2f,0x2c,0x29,0x2a,
    0x0b,0x08,0x0d,0x0e,0x07,0x04,0x01,0x02,0x13,0x10,0x15,0x16,0x1f,0x1c,0x19,0x1a};

//This function does MixColumns on a single column
int mixColumn (int word){

    int j = 0;
    int temp;
    uint8_t bytes[4];
    uint8_t mixedBytes[4];


    for (j = 0; j < 4; j++){
        bytes[3-j] = ((word>>(8*j))&0xFF);
    }
        mixedBytes[0] = galois2[bytes[0]]^galois3[bytes[1]]^bytes[2]^bytes[3];
        mixedBytes[1] = bytes[0]^galois2[bytes[1]]^galois3[bytes[2]]^bytes[3];
        mixedBytes[2] = bytes[0]^bytes[1]^galois2[bytes[2]]^galois3[bytes[3]];
```

```
        mixedBytes[3] = galois3[bytes[0]]^bytes[1]^bytes[2]^galois2[bytes[3]];

        temp = mixedBytes[0] << 24|mixedBytes[1] << 16|mixedBytes[2] << 8|mixedBytes[3];
        return temp;
}

int main(){

//Variable declaration
    uint32_t x1, x2, x3, x4;
    uint32_t fx1, fx2, fx3, fx4;
    uint32_t A1, A2, A3, A4;
    int i, j, k, l, S, R;
    int temp;
    int eprime2 [4096];
    int *ptr;
    int *pointers[2]; //size must be changed to use more than 2 faulty ciphers
    int cnt = 0;
    int counts[2];
    int size = 0;


//Determine all contents of e': e' = MixColumns(e) - Equation 3.6

    for (i = 0; i<4; i++){
        for (j = 0; j<256; j++){
            temp = mixColumn((j<<8*i));
            eprime2[1024*i + 4*j] = (temp>>24)&0xFF;
            eprime2[1024*i + 4*j + 1] = (temp>>16)&0xFF;
            eprime2[1024*i + 4*j + 2] = (temp>>8)&0xFF;
            eprime2[1024*i + 4*j + 3] = temp&0xFF;
        }
    }

//IO functions
    printf("Enter the 4 correct bytes of the ciphertext in hex, separated by ENTER\n");
    scanf("%x", &x1);
    scanf("%x", &x2);
    scanf("%x", &x3);
    scanf("%x", &x4);

    for(R = 0; R<(sizeof(pointers)/4); R++){
        cnt = 0;
        ptr = (int*) malloc(1024*sizeof(int));
        size = 1024;
        printf("Enter 4 faulty bytes of the ciphertext in hex, separated by ENTER\n");
```

```
        scanf("%x", &fx1);
        scanf("%x", &fx2);
        scanf("%x", &fx3);
        scanf("%x", &fx4);

        A1 = x1^fx1;
        A2 = x2^fx2;
        A3 = x3^fx3;
        A4 = x4^fx4;


//Byte-for-byte, check if Equation 3.6 is satisfied for all candidates for I,
//and if so, keep the candidate
        for(S = 0; S<1024; S++){
            for(i=0;i<256;i++){
                temp = sbox[i]^sbox[i^eprime2[4*S]];
                if(temp == A1){
                    for(j=0;j<256;j++){
                        temp = sbox[j]^sbox[j^eprime2[4*S + 1]];
                        if(temp == A2){
                            for(k=0;k<256;k++){
                                temp = sbox[k]^sbox[k^eprime2[4*S + 2]];
                                if (temp == A3){
                                    for(l=0;l<256;l++){
                                        temp = sbox[l]^sbox[l^eprime2[4*S + 3]];
                                        if(temp == A4){
                                            //printf("%02x%02x%02x%02x ", i, j, k, l);
                                            cnt=cnt+1;
                                            //printf("%d\n", cnt);
                                            if(cnt>=size){
                                                size = size*2;
                                                ptr = (int*) realloc(ptr, size*sizeof(int));
                                                if(ptr == NULL){
                                                    printf("Error! memory not allocated.");
                                                    exit(0);
                                                }
                                            }
                                            *(ptr + cnt) = ((i<<24) | (j<<16) | (k<<8) | l);
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
```

```
        }
        pointers[R] = ptr;
        counts[R] = cnt;
    }


//If D1 and D2 give an overlapping candidate, print. This part must be modified if
//you want to use more faulty ciphers
    for(i = 0; i<counts[0]; i++){
        for(j = 0; j<counts[1]; j++){
            if(*(pointers[0]+i+1)==*(pointers[1]+j+1)){
                printf("%08x\n", *(pointers[0]+i+1));
            }
        }
    }

    return 0;
}
```

A.3 Inverse key expansion

The C code below performs the AES Key Schedule in inverse, allowing for the retrieval of the secret key if K^{10} is known. Compile it, run it, and enter K^{10}. For example, entering B4EF5BCB-3E92E211-23E951CF-6F8F188E will yield the key of all-zeroes used in Chapter 6.

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

//Round Constants
static const uint32_t rcon[10] = {
    0x01000000, 0x02000000, 0x04000000, 0x08000000, 0x10000000,
    0x20000000, 0x40000000, 0x80000000, 0x1b000000, 0x36000000};

//The AES S-box
static const uint8_t sbox[256] = {
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16 };

//IO function
void keyInput (int *keys){
    printf("Enter the 4 words of K10 in hex, separated by ENTER \n");
    scanf("%x", (keys + 40));
    scanf("%x", (keys + 41));
    scanf("%x", (keys + 42));
    scanf("%x", (keys + 43));
}

//eShiftRow performs a 1-byte left-shift on a word, similar to shiftRows
uint32_t eShiftRow(uint32_t row){ //FOR USE IN KEY EXPANSION
```

```
    uint8_t a, b, c, d;
    uint32_t temp;

    a=(row&0xFF); //extract first byte
    b=((row>>8)&0xFF); //extract second byte
    c=((row>>16)&0xFF); //extract third byte
    d=((row>>24)&0xFF); //extract fourth byte

    temp = c << 24|b << 16 | a << 8 | d;
    return temp;
}

//SubWord applies the S-box to all 4 bytes of its input argument
uint32_t subWord(uint32_t word){

    uint8_t a, b, c, d;
    uint32_t temp;

    a = (word&0xFF); //extract first byte
    b = ((word>>8)&0xFF); //extract second byte
    c = ((word>>16)&0xFF); //extract third byte
    d = ((word>>24)&0xFF); //extract fourth byte

    a = sbox[a];
    b = sbox[b];
    c = sbox[c];
    d = sbox[d];

    temp = d << 24|c << 16|b << 8|a;
    return temp;
}

int main(){

//variable declaration, including an array to store the round keys
    uint32_t keys[11][4] = {
    0x00000000, 0x00000000, 0x00000000, 0x00000000,  //KEY
    0x00000000, 0x00000000, 0x00000000, 0x00000000,  // K1
    0x00000000, 0x00000000, 0x00000000, 0x00000000,  // K2
    0x00000000, 0x00000000, 0x00000000, 0x00000000,  // K3
    0x00000000, 0x00000000, 0x00000000, 0x00000000,  // K4
    0x00000000, 0x00000000, 0x00000000, 0x00000000,  // K5
    0x00000000, 0x00000000, 0x00000000, 0x00000000,  // K6
    0x00000000, 0x00000000, 0x00000000, 0x00000000,  // K7
    0x00000000, 0x00000000, 0x00000000, 0x00000000,  // K8
```

```
    0x00000000, 0x00000000, 0x00000000, 0x00000000,   // K9
    0x00000000, 0x00000000, 0x00000000, 0x00000000}; // K10

    int i = 0, j = 0;

//IO Function
    keyInput(&keys[0][0]);

//Perform the inverse key schedule
    for (i = 9; i > -1; i--){
        for (j = 3; j > -1; j--){
            if (j == 0){
                keys[i][j] = keys[i+1][j]^subWord(eShiftRow(keys[i][j+3]))^rcon[i];
            }
            else{
                //printf("%08x %08x \n", keys[i+1], keys[])
                keys[i][j] = keys[i+1][j]^keys[i+1][j-1];
            }
        }
    }
//Print the Key
    printf("%08x %08x %08x %08x \n", keys[9][0], keys[9][1], keys[9][2], keys[9][3]);
    printf("The original key is:\n");
    printf("%08x %08x %08x %08x", keys[0][0], keys[0][1], keys[0][2], keys[0][3]);
    return 0;
}
```

B VHDL TMR Implementation

In this appendix the code for the TMR implementation from Chapter 5 is presented.

B.1 Top-level TMR description

The VHDL description below is for the top-level of the TMR implementation presented in Figure 5.1.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY AES_TMR IS
        PORT (CLK                       : IN  STD_LOGIC;
              RESET             : IN  STD_LOGIC;
                    -- CONTROL PORTS ------------------------------------
          DATA_AVAIL      : IN  STD_LOGIC;
          DATA_READY      : OUT STD_LOGIC;
                    -- DATA PORTS ---------------------------------------
          KEY                       : IN  STD_LOGIC_VECTOR (127 DOWNTO 0);
          DATA_IN             : IN  STD_LOGIC_VECTOR (127 DOWNTO 0);
          DATA_OUT            : OUT STD_LOGIC_VECTOR (127 DOWNTO 0)
          );
END AES_TMR;

ARCHITECTURE Structural OF AES_TMR IS

SIGNAL DATA_1, DATA_2, DATA_3       : STD_LOGIC_VECTOR ( 127 DOWNTO 0);
SIGNAL READY_1, READY_2, READY_3            : STD_LOGIC;

BEGIN
        AES_1 : ENTITY work.AES_Core
        PORT MAP (CLK => CLK, RESET => RESET, DATA_AVAIL => DATA_AVAIL, KEY => KEY,
        DATA_IN => DATA_IN, DATA_OUT => DATA_1, DATA_READY => READY_1);

        AES_2 : ENTITY work.AES_Core
        PORT MAP (CLK => CLK, RESET => RESET, DATA_AVAIL => DATA_AVAIL, KEY => KEY,
        DATA_IN => DATA_IN, DATA_OUT => DATA_2, DATA_READY => READY_2);

        AES_3 : ENTITY work.AES_Core
        PORT MAP (CLK => CLK, RESET => RESET, DATA_AVAIL => DATA_AVAIL, KEY => KEY,
        DATA_IN => DATA_IN, DATA_OUT => DATA_3, DATA_READY => READY_3);

        Voter : ENTITY work.AES_Voter
```

```
        PORT MAP (DATA_1 => DATA_1, DATA_2 => DATA_2, DATA_3 => DATA_3,
        READY_1 => READY_1, READY_2 => READY_2, READY_3 => READY_3,
        DATA_OUT => DATA_OUT, READY_OUT => DATA_READY);
END Structural;
```

B.2 Voter

The VHDL description below is for the voter presented in Figure 5.2.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY AES_Voter IS
        PORT (
            -- DATA PORTS ------------------------------------------
               DATA_1          : IN  STD_LOGIC_VECTOR ( 127 DOWNTO 0);
               DATA_2          : IN  STD_LOGIC_VECTOR ( 127 DOWNTO 0);
               DATA_3          : IN  STD_LOGIC_VECTOR ( 127 DOWNTO 0);
            -- CONTROL PORTS ---------------------------------------
               READY_1         : IN  STD_LOGIC;
               READY_2         : IN  STD_LOGIC;
               READY_3         : IN  STD_LOGIC;
            -- OUTPUT PORTS ----------------------------------------
               DATA_OUT        : OUT STD_LOGIC_VECTOR ( 127 DOWNTO 0);
               READY_OUT       : OUT STD_LOGIC
             );

END AES_Voter;

ARCHITECTURE Behavioral OF AES_Voter IS

BEGIN

        READY_OUT <= READY_1 AND READY_2 AND READY_3;

        DATA_OUT <= DATA_1 when DATA_1 = DATA_2 AND DATA_2 = DATA_3 else (others => '0');

END Behavioral;
```