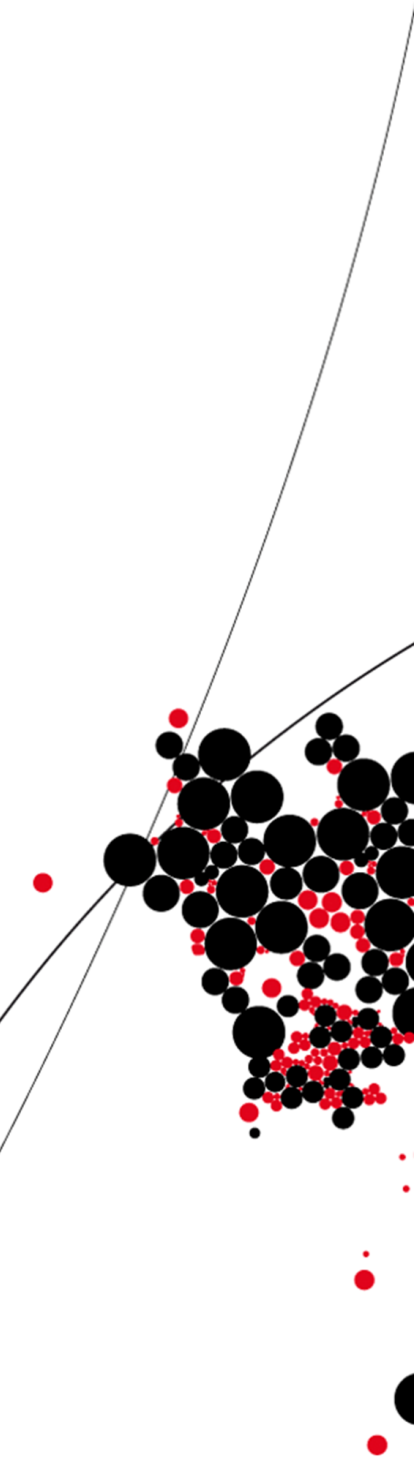
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Analysing Countermeasures against Fault Injection Attacks on FPGA based Cryptographic Implementations

T.R. Haarman

B.Sc. Thesis
June 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

Nowadays, *Field Programmable Gate Arrays* (FPGAs) are used for a very diverse range of applications, many of which are security-critical. As is often the case with security-critical systems, there are adversaries that develop ways to acquire the protected information from the system. And while the adversaries come up with increasingly advanced methods for succeeding in this, the countermeasures that are developed against them have to match or stay ahead of those attacks. On hardware, it's possible for an adversary to perform a *Fault Injection Attack* (FIA) which could present faulty cipher texts at the output. With these faulty cipher texts, it's possible to decrypt the original key by executing a *Differential Fault Analysis* (DFA). In this work, the aim is to analyse a state of the art countermeasure that combines protection against both fault injection and side channel attacks. First, methods to perform a successful FIA on the countermeasure are examined in a software implementation of an encryption algorithm protected with ParTi. When a method is developed, a modified version of the EDA toolkit DAVOS is used to perform multiple schemed FIAs on the hardware implementation. It is presented how the secret key can be extracted, and the rate of successful fault injections is determined.

Contents

1 Introduction

Nowadays, embedded systems are becoming more and more integrated in society. While in the beginning embedded computer systems were exclusively used by a confined group of users, currently a day without interacting with such a system seems almost impossible. FPGAs are widely used in embedded systems, because they are very flexible (programmable in the field) and efficient. Owing to their widespread use, FPGAs now find applications in process automation and communication systems, some of them dealing with essential and/or very personal data. If this data gets intercepted by adversaries it can be used to commit crimes with unforeseeable consequences, for example identity theft. To prevent adversaries from being able to extract valuable information from these devices, various safety measures have been developed, both for hardware and software. AES encryption [1] is often used to protect the data stream. But as FPGAs are often deployed in non-confined, sometimes even public places, adversaries can monitor the behaviour of a device and can possibly even edit encrypted data by means of a *Fault Injection Attack* (FIA) to recover the used key, which enables them to decrypt the secret data. To be prepared against this attacks, various countermeasures have been developed, of which *ParTI* [2] is regarded as the state of the art countermeasure. In this work, a modified version of the EDA toolkit *DAVOS* will be used to emulate FIAs on an FPGA running an AES implementation which is protected by ParTI. This way the probability of a successful FIA can be determined and recommendations can be made for improving the countermeasure.

This thesis will cover the use of cryptographic implementations on FPGAs with a focus on *Round-Based* AES in Chapter 2. Then in Chapter 3 the basic operations of ParTI against FIAs and *Side Channel Attacks* (SCAs) will be explained. Next, in Chapter 4, methods for extracting an AES key by using fault injection will be shown. In Chapter 5 the methods which are going to be used to inject faults are presented. Then, in Chapter 6, the vulnerabilities of the ParTI implementation on an FPGA will be determined using the previously described methods. In Chapter 7, the used methods are evaluated. Finally, in Chapter 8, the conclusions from this thesis are stated and recommendations are made to improve the safety of the current countermeasure.

2 Cryptographic Implementations

To encrypt secret data, various algorithms have been developed, which all have pros and cons. For this reason, the *National Institute of Standards and Technology* [3] chooses a standard for encryption, which they deem safe and usable for most general cases where encryption is needed. At the moment this is the *Advanced Encryption Standard* (AES) [1] which superseded the old *Data Encyrption Standard* (DES) in May 2002. Since becoming a standard, a variety of AES implementations have been proposed for FPGA based systems which require encryption.

2.1 Round-Based AES

AES contains three members of the Rijndael block cipher family, namely one with a key of 128, 192 and 256 bits. The three versions are called AES-128, AES-192 and AES-256 which take respectively 10, 12 and 14 rounds to execute. AES always takes the input data by blocks of 128 bits, as this is specified in the standard, while the Rijndael cipher can also process larger input data blocks. For explanation purposes, the focus in this thesis will be on AES-128 (from now on referred to as AES), as it is the most compact version.
In general, as AES has a substitution-permutation network, every round of the algorithm consists of substitution and permutation steps which are executed on the intermediate 128 bits "state". These steps give AES both the confusion and diffusion that is necessary for a strong cryptographic algorithm, according to Shannon [4]. Also, in every round, an expanded version of the key is added to the state by a bit-wise XOR operation. There are four steps per normal round, *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. In the last round of AES, MixColumns is not executed. To get an overview of the structure of the algorithm, take a look at Figure 2.1. In the next sections, an explanation of the main steps of the AES algorithm can be found. A more extensive technical description of the whole AES standard can be found in [1].

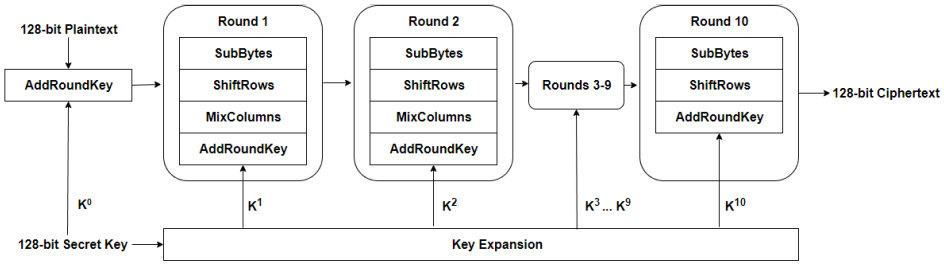


Figure 2.1: Structure of AES [5]

2.2 SubBytes

In *SubBytes*, every byte from the 16 byte state is replaced by another value. These values can be found in the S-box, that is generated by taking the multiplicative inverse of Galois Field $GF(2^8)$, which makes the S-Box very non-linear. The function of *SubBytes* is to add confusion to the algorithm. The S-box should be as non linear as possible to guarantee the most powerful protection against adversaries. The creators of the Rijndael cipher noted that AES could also work with other S-boxes as long as they fulfill the criteria defined in [6]. In Figure 2.2, a graphical representation of the *SubBytes* operation can be seen.

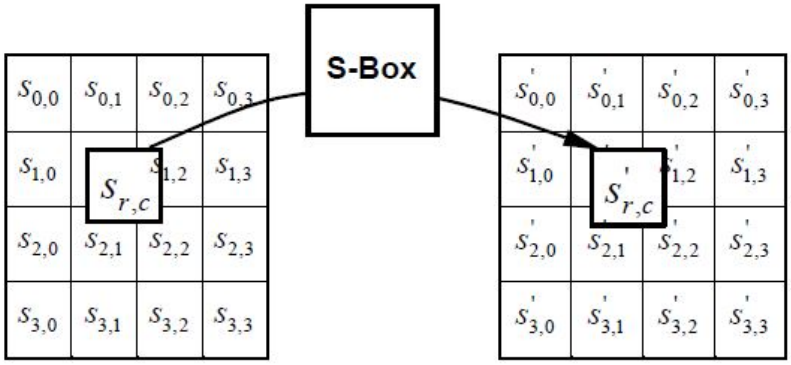


Figure 2.2: A graphical representation of SubBytes [1]

2.3 ShiftRows

As the name already suggests, during *ShiftRows* the state's values are shifted in the rows. A cyclical left-shift is executed on the rows. For the first row, no shift is performed, and then for the next rows the byte is shifted respectively 1, 2 and 3 positions, while the left most (lowest) position is shifted to the right (highest) position for every shift. By shifting around values in their rows this step introduces diffusion to the algorithm. In Figure 2.3, a graphical representation of the *ShiftRows* operation can be seen.

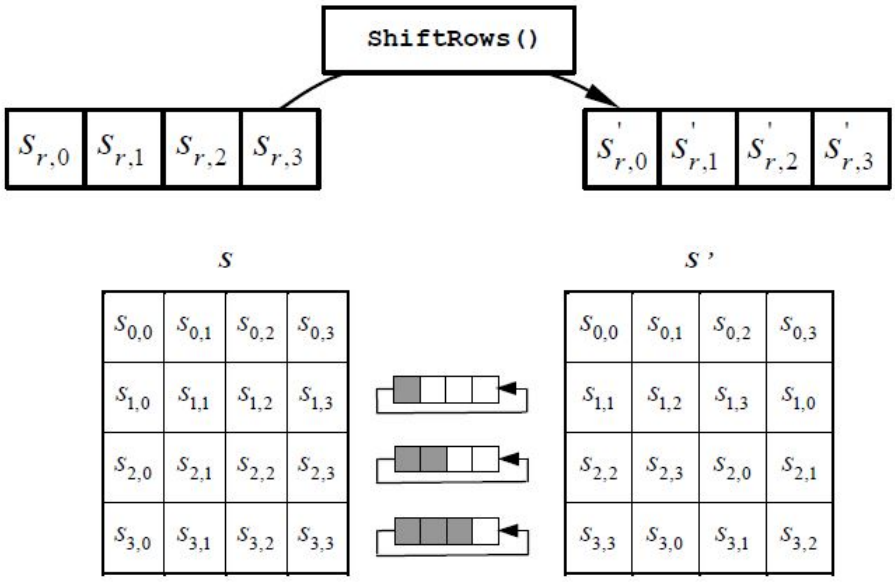


Figure 2.3: A graphical representation of ShiftRows [1]

2.4 MixColumns

The second diffusion step is *MixColumns*. This operation takes every column as a four term polynomial in the Galois Field $GF(2^8)$ and multiplies it modulo $x^4 + 1$ with a fixed polynomial $a(x) = 3x^3 + x^2 + x + 2$. As it is a multiplication in $GF(2^8)$ it can be considered as a matrix multiplication $s'(x) = a(x) \otimes s(x)$, this leads to (2.1) for every column.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \tag{2.1}$$

A graphical representation of *MixColumns* can be seen in Figure 2.4.

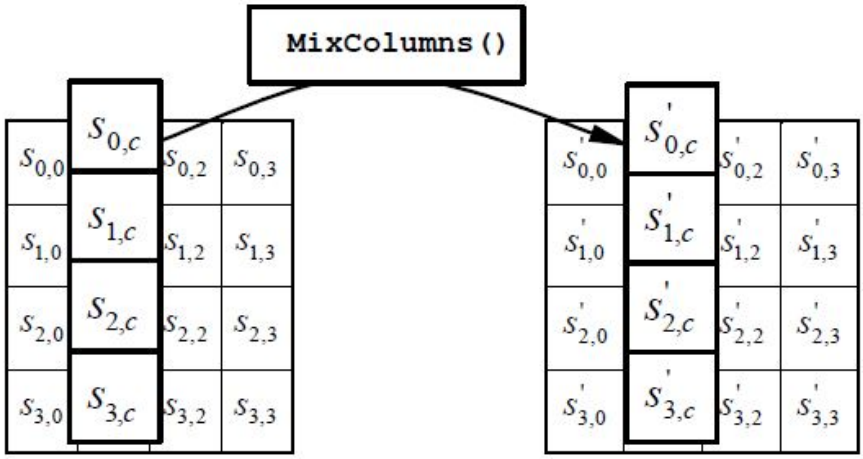


Figure 2.4: A graphical representation of MixColumns [1]

2.5 AddRoundKey

The last step in each round is adding the round key to the state by means of an XOR operation. In AES a round key schedule is generated from the original 128-bit key by means of a key expansion, which means that for every round a different, unique key is generated and added to the state. The key expansion scheme performs some operations on the previous round key to generate a new one where it also uses a specific array with round constants *rcon*. A graphical representation of *AddRoundKey* can be seen in Figure 2.5.

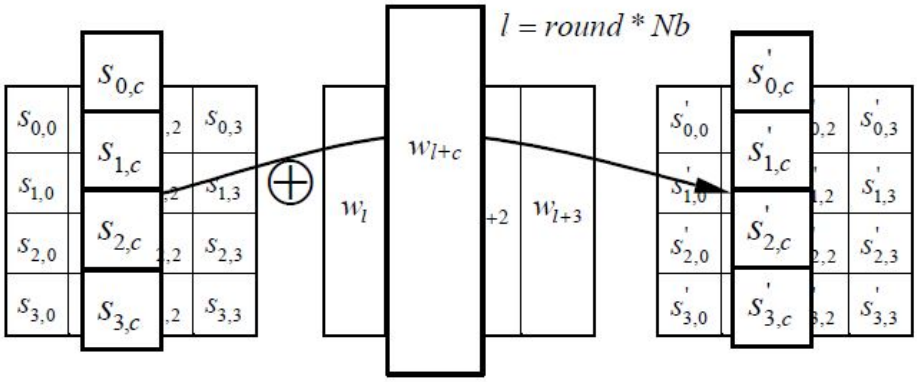


Figure 2.5: A graphical representation of AddRoundKey [1]

3 The ParTI Countermeasure

ParTI [2] is a countermeasure that is developed to provide combined protection against *Fault Injection Attacks* (FIAs) and *Side Channel Attacks* (SCAs). This combined protection makes it a unique solution, as most countermeasures are just aimed at protection against one of the two types of attacks. Often two different countermeasures cannot be combined without sacrificing some level of protection of the other countermeasure. This is especially the case when using more sophisticated FI protection schemes on channels that are protected against SCAs, as they often require some unmasked parity checks, which makes the device susceptible for SCAs again.
ParTI combines two extensively researched countermeasures to function together, while claiming to keep a high level of protection against both types of attacks. The protection against SCAs is based on *Threshold Implementations* [7] and the protection against FIs is based on *Error Detecting Codes* [8]. Those two concepts will be described in Sections 3.1 and 3.2.

3.1 Threshold Implementations

Differential Power Analysis (DPA) [9] is one of the most practical SCAs in which the adversary measures the power usage of a specific part of the FPGA, which enables the adversary to analyse the power consumption behaviour of the cryptographic implementation on the board. One way to prevent successful DPA is by masking the channels on the hardware, so effectively hiding the real power consumption by randomizing it. One way to do this is with *Threshold Implementations* (TIs). In this thesis the general principle is highlighted, a more in depth coverage can be found in [10] and [11].
A TI is essentially a masking scheme for data channels on an FPGA. It splits the information in various shares, which are all non-complete, but when combined in the end lead to the right output again. As large, high power consuming processes are now split into smaller shares with a more equal power consumption, a uniform distribution of resource usage can be achieved, which makes SCAs like DPA more difficult.
To demonstrate the basic functioning of a TI, an example of a 4 bit intermediate value can be taken. Let's represent this value as $\mathbf{x} =< x_1, ..., x_4 >$, which will be represented in TI as a Boolean masked form of the order (n-1), so as $(\mathbf{x_1}, ..., \mathbf{x_n})$. The original $\mathbf{x}$ can then be constructed by $\mathbf{x} = \bigoplus_{i=1}^{n} \mathbf{x_i}$, where every share is a 4 bit value on it's own, so $\mathbf{x_i} =< x_1^i, ..., x_4^i >$.
When implementing this on for example AES, there are linear steps (like the *ShiftRows* and *MixColumns* functions) and non-linear steps (most importantly the S-Box). For the linear steps it is easy to use them with a TI, because of the principles of linearity allowing to replace the one value with all smaller shares leading to this same value together. So, $ShiftRows(\mathbf{x}) = \bigoplus_{i=1}^{n}(ShiftRows(\mathbf{x_i}))$

For the non-linear operations, it is more difficult to keep the uniformity of the shares in place, while also preserving the bijective behaviour of the S-Box. In [11], a complete TI for the AES S-Box can be found. Here the S-Box operation is split in 6 stages, which are separated by registers. This is done to keep the non-completeness property true within each stage.

3.2 Error Detection

ParTI combines the principle of *Error Detecting Codes* (EDCs) with *Concurrent Error Detection* (CED) [12]. EDCs can be used to detect and even correct faults that occur on an unreliable communication channel. To achieve this it uses a generator matrix to make codewords which consists of the original input (codeword) padded with CheckBits, which can later be used to check the correctness of the codeword. The definitions for the EDCs that are used in ParTI can be found in [2].
Concurrent Error Detection typically uses redundancy to detect faults during runtime. In the simple case this just means running multiple executions in parallel or sequential, which increases resource usage a lot. In ParTI, the CED scheme is combined with EDCs, to be able to detect and avert more faults. In Figure 3.1 a basic implementation of this combination can be found. Here, it can be seen that a specific operation is being protected by CED while generating *CheckBits* before and then passing this value through a predictor of the operation. The Predictor block is performing the same operation as the Operation block, only on the data which is coded with *CheckBits* After this, there is an error check, where the output of the normal operation is coded with *CheckBits* and then compared with the output of the Predictor. If the *CheckBits* don't match, an error will be detected and the Error state will be set.

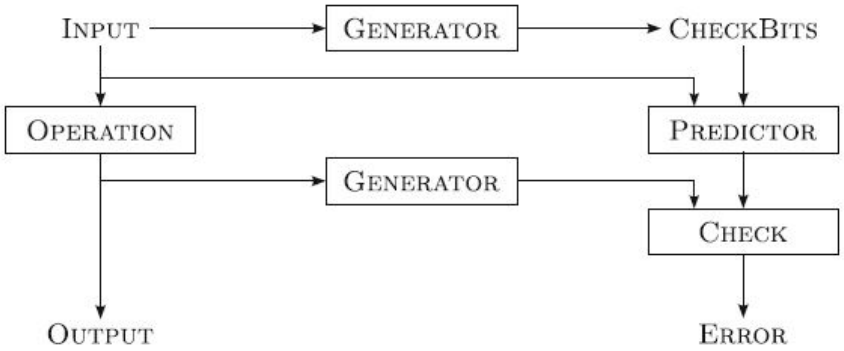


Figure 3.1: A basic implementation of CED with EDCs [2]

3.3 A Combined Scheme

To get a safe combined scheme for SCAs and FIAs, it's important to make all the parts of the scheme work according to the principles of TI, also the predictors. To keep this condition true, also the EDCs have to be included in the TI design. A systematic linear code is used, which makes it possible to also correct possible errors that occur.
The code of *ParTI* is constructed in a way that the computations of the target algorithm (for example AES) and the predictors can be split, and executed next to each other. This means that both outputs are calculated completely independent from each other. The basic scheme of *ParTI* can be found in Figure 3.2.

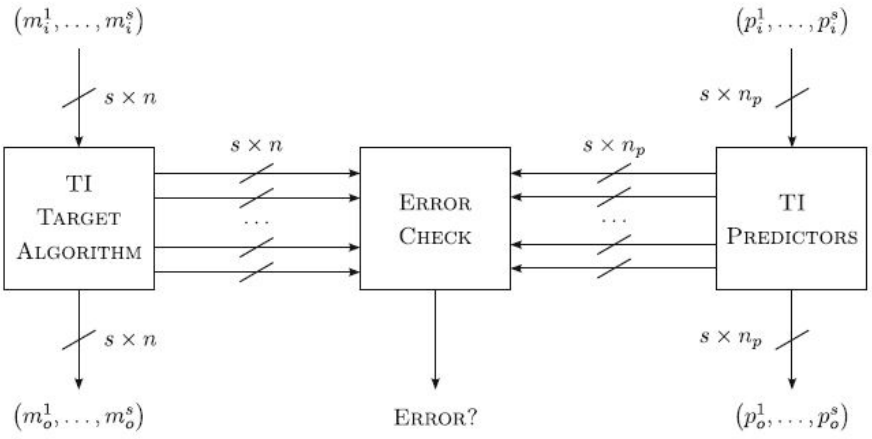


Figure 3.2: The general implementation of ParTI [2]

In Figure 3.2 it can be clearly seen that the target algorithm and predictors are executed separated from each other, with one exception, which is the error checking part. m_i is the not encoded AES state which consists of s separate shares of bit length n. p_i is the encoded version of the input which consists of s separate shares of bit length n_p. The error check is the part where intermediate states from both sides are compared, and which indicates when it detects an error. The number of intermediate checks can be different for each design, but it's important to keep in mind that while increasing the security, having more error checks also has a negative effect on performance and area consumption on the hardware. So it is important to find the right balance for this.

4 Differential Fault Analysis

Multiple methods have been developed to retrieve the original key from an AES implementation. One of the most promising methods at the moment is Differential Fault Analysis (DFA), where a fault is injected into a cryptographic implementation to generate faulty ciphers which can be used to restore the state of the key in a certain round and then simply retrieve the original key using the key scheme. This method was proven to be successful on an unprotected round-based AES implementation by Giraud in 2005 [13]. Later a more generalized approach was worked out by Moradi, Shalmani and Salmasizadeh in 2006 [14]. In the next two sections, both concepts are reviewed.

4.1 Giraud attack

In the Giraud attack, a single bit error is inserted at the input of the 10th round of *SubBytes*. By only altering it before the last round of AES, there will be no MixColumns operation that spreads the error over multiple bytes, the fault will just propagate to one byte, as can be seen in Figure 4.1. The attack scheme of a Giraud attack can be found in [5]. With one bit faults, under 50 faulty bytes (3 per byte) are necessary for retrieving the key of AES-128 [13].

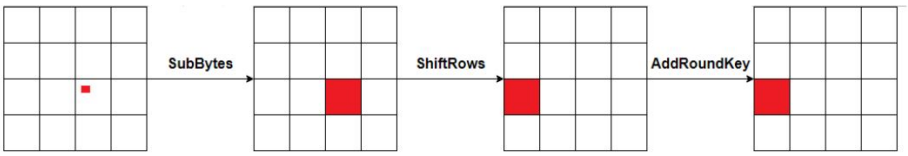


Figure 4.1: The propagation of a fault inserted before the 10th round of SubBytes [5]

4.2 Generalized attack on AES

To be able to retrieve the original key from a situation when inserting multiple bit faults, a more general attack at the input of the 9th round of *SubBytes* is possible [14]. One of the proposed attacks in this work can deal with 1, 2 or 3 bytes containing one or more bit errors in one word at the input of the 9th round of *SubBytes*, which makes it possible to retrieve the original key with just 6 faulty ciphers. The great advantage of this kind of attack is that it is more general and able to deal with multiple flipped bits in a byte. A detailed scheme of this attack can be found in [5] and in Figure 4.2, the propagation of a fault induced in a single byte at the input of the 9th round of *SubBytes* can be seen.

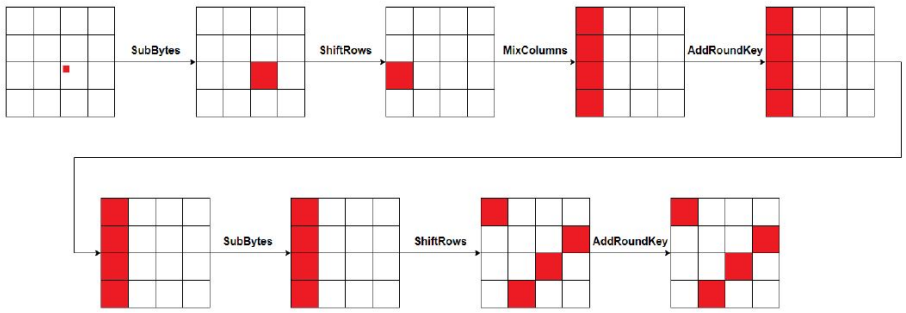


Figure 4.2: The propagation of a fault inserted before the 9th round of SubBytes [5]

5 Fault injection methods

Multiple steps need to be taken to get a good understanding of the response of ParTI to fault injections and to set up an attack scheme to check ParTIs fault correction and detection capabilities on hardware. First, a C emulation is made for efficient realization of many situations. Then a method to inject faults on hardware bitstreams during runtime has to be used and the error detection rates have to be determined.

5.1 C Emulation of ParTI

The first stage in testing the fault tolerance of ParTI is done by programming a C emulation of it. A C emulation is used because in a language like C it is very easy to make the functioning and structure of ParTI visible. Besides that it is more time efficient than testing out various options on a hardware design program, as they take a lot of time for the various code generation steps in contrast to the quick compilation speed of a regular C code.
The application was set up to resemble the structure of the original hardware implementation as much as possible. An overview of the structure of a regular round as implemented in the C application can be seen in Figure 5.1. In the figure, the CEC functions are the predictors of the operations they are named after, they work with the encoded version of the input, which contain the *CheckBits*. In the correction stage it can be seen that both flows are combined to check for errors by using *CheckBits*. When an error is detected, this function attempts to correct faults, and if that is not possible it will set the outputs to undefined. The different stages are controlled by the AES() function, which is designed to replicate the AES Core functionality as in the VHDL version of ParTI as much as possible. The C code can be found in Appendix A.

To determine the success of an attack, the AES() function returns a flag to indicate if there was an error detected during the runtime. Also the correctness of the output is checked, and if the output is not correct, it is checked if the error was detected. If the error was detected it would mean on the hardware implementation that the output would have been undefined, so not usable for an adversary. If the the output is incorrect and the error was not detected, it means that the adversary was able to change the output and could be able to crack the cipher.
From the acquired results, the expectations for the fault percentages are presented and can later be used to compare with the results on hardware.

5.2 Specifying the attack scheme

To perform a successful attack, a scheme has to be set up which takes into account properties of ParTI while combining them with the prerequisites for the type of attack that is chosen. This attack scheme is based on the

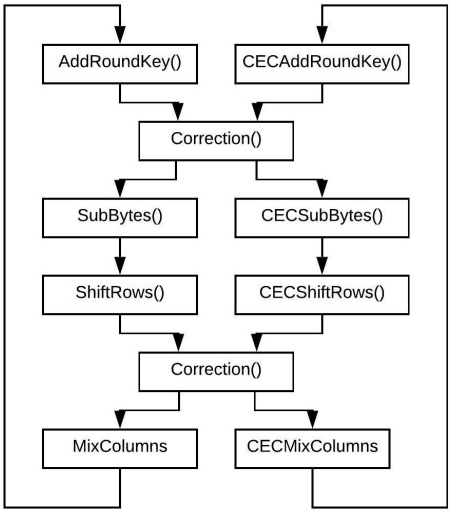


Figure 5.1: A regular round with ParTI implemented in C

attacker model that ParTI is based on, which for example says that the adversary can only inject faults in the data path and not in the control flow [2].
Furthermore, the faults have to be injected at a specific time and location. These are dependent on the type of attack that is used. It is known that the Concurrent Error Correction (CEC) in *ParTI* can successfully avert all faults with a multiplicity 1 (one bit error), hence, the Giraud attack for single bit errors will not be suitable. The generalized DFA attack working with one faulty byte at the input of the 9th round of *SubBytes* can be used though, as faults with a multiplicity of 2 or higher will affect the cipher output.
To summarize, faults with a multiplicity of 2 or higher have to be inserted in a single byte at the input of the 9th round of *SubBytes*, which will result in 4 faulty bytes at the output cipher which can then be used to recover the original key using the generalized DFA method specified in [14]. The adversary is assumed to be able to inject faults with a multiplicity higher than 1, which are uniformly distributed over one specified region at a specific time instance.

5.3 Injecting Faults with DAVOS

To inject faults into the bitstream on an FPGA, the *Electronic Design Automation* (EDA) toolkit *DAVOS* can be used. This toolkit was in principle developed for dependability assessment, verification, optimization and selection of hardware models [15]. It is built up from separate custom modules for different steps in the process. One of the modules is the *fault injection module* (FFI), which is originally meant for dependability testing, but can also be used to emulate fault injection attacks by altering the bitstream. In [5], *DAVOS* is used to insert single bit faults, which ParTI should be able to correct in any case as mentioned in [2]. So to be able to achieve successful fault propagation on a *ParTI* implementation, *DAVOS* should be configured in a way that it can insert multiple bit faults in one clock cycle. Besides this, a relevant attack scheme as specified in Section 5.2 should be implemented in *DAVOS* and needs to have a similar setup as the one that was used in the C emulation, to be able to compare the results.
The configuration of the test setup and the faults that will be injected is set up in an XML test configuration file and in the Python code of the fault injection tool. Here, it can be specified, amongst others, on which clock cycle an attack has to be executed, how much faults are inserted, in which kind of logic those errors are inserted, what is the type of fault that is emulated. To be able to inject multiple faults in one clock cycle, the fault multiplicity has to be set. The fault multiplicity is defined as the number of faults per experiment over the number of gates in the circuit [16]. For example, when an injection with multiplicity 2 in a buffer of 128 bits (16 bytes) is executed, there will be 2 random bits altered over the whole buffer.

The hardware that is used in this thesis to perform the fault injection experiments with *DAVOS* is the *ZedBoard* [17], which is equipped with an xc7z020 device.

5.4 Determining fault coverage

To determine the fault coverage rate of *ParTI* for different fault multiplicities, assuming the faulty bits are distributed uniformly, the minimal number of fault injection experiments that have to be executed to theoretically perform all possible injections is equal to the number of combinations $\binom{N}{k}$. Here N corresponds to the number of bits in the targeted logic and k corresponds to the fault multiplicity. This would mean that the number of experiments needed to get an indication of all possible outcomes climbs rapidly with a rising multiplicity and/or block size, as can be seen in Table 5.1.

N\k	1	2	3	4
128	128	8,128	341,376	10,668,000
192	192	18,336	1,161,280	54,870,480
256	256	32,640	2,763,520	174,792,640

Table 5.1: Number of experiments needed for various block sizes and fault multiplicities

However, there is a limited number of faulty values required per byte to successfully retrieve the original key when performing a generalized DFA attack. So not all theoretical combinations have to be inserted to the system to be able to perform a successful DFA.

6 Results

In this section, the results of fault injection on round-based AES protected by the *ParTI* countermeasure can be found. First, predictions for the fault coverage rate are made with the C emulation. Next, the results for fault injection on the *ZedBoard* using *DAVOS* are shown.

6.1 Fault injections to the emulation

First, to test the functioning of the emulation the claim that all single bit flips are corrected is tested by inserting one bit errors in every possible round input of *SubBytes*, which indeed leads to only correct outputs. As described in Section 5.2, a generalized DFA attack is deemed a suitable method for potentially retrieving the original key from round-based AES protected by *ParTI*. The attack is first performed in the emulator to get an indication of the propagation of faults. For every injection, just 2 bits are flipped within one byte at the input of the 9th round of *SubBytes*. This is done to meet the minimum of a multiplicity of 2 that is necessary for a fault to propagate to the output and for the DFA results to be interpreted easily.
In Table 6.1, the results from this injection on the emulation can be found for all possible combinations of two flipped bits within one byte at the input of the 9th round of *SubBytes*, according to the code in Appendix A. It is important to realize that this just indicates the effectiveness of very biased attacks which are focused on one byte within the input.

N experiments	448	100%
Correct Outputs	280	62.5%
Detected Errors	90	20.1%
Undetected Errors	78	17.4%

Table 6.1: 2 bit fault injections at the input of 9th round SubBytes of the emulation

The results from Table 6.1 show that just 17.4% of the injected faults propagate to the output, even when targeting single bytes in a very biased attack. When an error is detected, on hardware it would mean that the output values would become undefined, which renders these outputs not useful for a DFA attack.

6.2 Fault injections to the bitstream

Now the *DAVOS* fault injection tool is used to insert faults into the bitstream on the *ZedBoard* FPGA target, the xc7z020 device. Again, as in Section 6.1, first attacks with a multiplicity of one are executed, at the input of every *SubBytes* round. Also on the hardware this leads to only correct outputs.
A big difference between the emulation and the fault injection on hardware is the bias of the attack. In emulation it was very easy to specify an exact position for injecting a fault. With the hardware injection, just the register in which the fault has to be inserted can be specified.
This means that even with a fault multiplicity of 2, a lot of times there will just be one faulty bit per byte, which will mean that the output is not affected. When choosing a multiplicity of 3, the chances that 2 bits are flipped within one byte are higher, while it is still not possible for multiple bytes to be changed, as that would require 4 flipped bytes (2 in one byte and 2 in another). For the correct timing of an injection, a timing sheet of the VHDL implementation of the finite state machine of round-based AES protected with *ParTI* has been made, which can be found in Appendix B. The settings that are used for the fault injection are summarized in Table 6.2.

Injection Round	9th
Injection Time	210 [ns]
Fault Multiplicity	3
Attack Scope	SubBytes Buffer
Number of injections	1028

Table 6.2: Settings for the DAVOS fault injection tool

Running this experiment with the settings from Table 6.2 gives 64 faults that propagate to the output, for 1028 injections, which yields an fault percentage of 6.23% for an attack with a multiplicity of 3.
From the acquired faulty ciphers, 2 have to be selected per word to perform a generalized DFA attack. The method proposed in [5] is now used to find the 4 input words of the tenth round of *SubBytes* (I) and then the 10th round key can be found per word using the correct cipher output C and I, according to equation 6.1.

$$K10_{0,13,10,7} = SubBytes(I) \oplus C_{0,13,10,7} \tag{6.1}$$

In Table 6.3 it is shown how the results of the fault injection are used to recover the original AES key, which is all zeros in this test case. The plain text that is used for encryption is {80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00}. D1 and D2 represent the faulty cipher bytes that are used to retrieve I as the input of the 10th round of *SubBytes*.

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
D1	**2D**	*14*	26	AD	*72*	23	AF	**CF**	74	BB	**BB**	*90*	26	**AB**	*8F*	49
D2	**D4**	*EB*	31	3A	*95*	36	FE	**A7**	0A	F7	**D2**	*5C*	3B	**75**	*49*	40
I	**E6**	**B9**	**9A**	**A2**	*48*	*76*	*BA*	*AE*	8D	F0	B5	C0	5D	B9	94	DB
K10	**B4**	*EF*	5B	CB	*3E*	92	E2	**11**	23	E9	**51**	*CF*	6F	**8F**	*18*	8E
K	**00**	*00*	00	00	*00*	00	00	**00**	00	00	**00**	*00*	00	**00**	*00*	00

Table 6.3: Successful recovering the AES key with generalized DFA from 8 faulty ciphers

7 Evaluation

As can be seen in the previous section, *DAVOS* was used to insert faults with a multiplicity higher than 1 to a protected round-based AES implementation. This makes it possible to emulate advanced adversaries who are in possession of equipment to insert multiple faults in the same clock cycle on a specific area.
Currently it is just possible for *DAVOS* to target e.g. a whole register, which means that the faults will still be uniformly distributed over this register. When targeting a system which is protected with a countermeasure like *ParTI*, this means that a high number of injections is necessary to get enough faulty cipher outputs. As adversaries are becoming more capable of setting up biased attacks, it would be a possible improvement to give the user more control over the bias of an emulated attack in *DAVOS*. This would greatly reduce the number of necessary injections while greatly extending the amount of possible fault injection attacks. A flexible range together with the configurable multiplicity would make it possible to emulate a very wide range of adversaries, from basic till very advanced.

It remains very important to realize that all the current attacks on *ParTI* have been performed while knowing the exact layout of the system. This could be very difficult for an adversary to know in practice, because of the Threshold Implementations which make identifying a specific area on the hardware particularly hard.

8 Conclusions and Recommendations

In this thesis, the resistance against fault injection attacks of a state of the art countermeasure, *ParTI*, has been analysed. This was done both in a software environment programmed in C and directly to the bitstream of an FPGA. For targeting the bitstream a module of the *DAVOS* toolkit was used, which is customized to perform FIAs.
Multiple types of attacks have been performed on the round-based AES protected by *ParTI*. First, attacks with a multiplicity of 1 on different rounds of AES confirmed the statement made in [2] that *ParTI* is capable of averting any single bit error which is injected in the data stream. Next, emulating fault injections with a higher multiplicity proved that when the adversary is capable of performing highly biased attacks, where at least 2 bits are altered within one data byte, faults will propagate to the output.
Then using a generalized DFA method which is able to deal with multi bit faults it was shown that the original key can be extracted from an implementation of round-based AES protected by the *ParTI* countermeasure.

Over all, it has been shown that, when an adversary is advanced enough to inject multiple faults in a specified area in one clock cycle and has a good knowledge of the layout of the implementation on the FPGA, it is possible to extract the original key. For now, this seems possible only in the lab in a very controlled environment. But as the technology that is available to adversaries is getting more and more advanced, it is of great importance to research possibilities to make countermeasures more resistant to higher multiplicity faults.
DAVOS as a fault injection platform can also be improved to give users more control of the bias of their emulated attacks, while also making the tool more user friendly.

Bibliography

[1] M. J. Dworkin, "Advanced encryption standard (AES)," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep., nov 2001. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[2] T. Schneider, A. Moradi, and T. Güneysu, "ParTI - Towards combined hardware countermeasures against side-channel and fault-injection attacks," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016.

[3] "National institute of standards and technology." [Online]. Available: https://www.nist.gov/

[4] C. E. Shannon, "Communication Theory of Secrecy Systems*," *Bell System Technical Journal*, vol. 28, no. 4, pp. 708–710, oct 1949. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6769090

[5] E. Van der Ploeg, "Developing a Platform to Emulate Fault Injection Attacks on Cryptographic Implementations," Ph.D. dissertation, University of Twente, 2020.

[6] J. Daemen and V. Rijmen, "The Rijndael Block Cipher: AES Proposal," in *Nist*, 2003.

[7] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2006.

[8] F. MacWilliams and N. Sloane, *The Theory of Error Correcting Codes*. New York: North- Holland Mathematical Library. North-Holland Publishing Co., 1977.

[9] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1999.

[10] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen, "Higher-order threshold implementations," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014.

[11] T. De Cnudde, B. Bilgin, O. Reparaz, V. Nikov, and S. Nikova, "Higher-order threshold implementation of the AES S-box," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016.

[12] X. Guo, D. Mukhopadhyay, C. Jin, and R. Karri, "Security analysis of concurrent error detection against differential fault analysis," *Journal of Cryptographic Engineering*, 2015.

[13] C. Giraud, "DFA on AES," in *Lecture Notes in Computer Science*, 2005.

[14] A. Moradi, M. T. Shalmani, and M. Salmasizadeh, "A generalized method of differential fault attack against AES cryptosystem," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2006.

[15] I. Tuzov, D. De Andres, and J. C. Ruiz, "DAVOS: EDA toolkit for dependability assessment, verification, optimisation and selection of hardware models," in *Proceedings - 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018*, 2018.

[16] M. Karpovsky, K. J. Kulikowski, and A. Taubin, "Differential fault analysis attack resistant architectures for the advanced encryption standard," in *IFIP Advances in Information and Communication Technology*, 2004.

[17] "ZedBoard." [Online]. Available: http://zedboard.org/product/zedboard

A C Emulation of ParTI

In this Appendix, the main part of C based ParTI emulation can be found, the complete source code can be found at: `https://drive.google.com/drive/folders/1oaqpsupHZb8ydFuJHrmpITOokKIuDwm2?usp=sharing`

A.1 main.c

```
#include "AES_Core.h"

static uint8_t plaintext[4][4] = {
    {0x80, 0x00, 0x00, 0x00},
    {0x00, 0x00, 0x00, 0x00},
    {0x00, 0x00, 0x00, 0x00},
    {0x00, 0x00, 0x00, 0x00}
};

static uint8_t ciphertext_check[4][4] = {
    {0x3a, 0x6c, 0x7e, 0x23},
    {0xd7, 0x1e, 0xbf, 0xd9},
    {0x8e, 0xc0, 0xe9, 0xec},
    {0x72, 0x2b, 0x2b, 0x34}
};

int main()
{
  int num_correct = 0;
  int num_wrong_detected = 0;
  int num_wrong_undetected = 0;

  //inserting 2 bits in the 9th round in every possible way for
    every byte.
  for(int error_round = 9; error_round <= 9; error_round++)
  {
    printf("error round: %d\n", error_round);
    for(int error_row = 0; error_row <= 3; error_row++)
    {
      printf("error row: %d\n", error_row);
      for(int error_col = 0; error_col <= 3; error_col++)
      {
        printf("error column: %d\n", error_col);
        for(int error_bit = 0; error_bit <= 7; error_bit++)
        {
          printf("first error bit: %d\n", error_bit);
          for(int error_bitdist = 1; error_bitdist <= (7 -
    error_bit); error_bitdist++)
          {
            uint8_t outputAES[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
            uint8_t outputCEC[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
            bool fault = 0;
            bool detected = 0;
```

```
42
43              //run a complete sequence of ParTI protected AES
44              detected = AES(plaintext, outputAES, outputCEC,
    error_round, error_col, error_row, error_bit, error_bitdist);
45
46              //checking the correctness of the output
47              for(int i = 0; i <= 3; i++)
48              {
49                for(int j = 0; j <= 3; j++)
50                {
51                  printf("%x ", outputAES[j][i]);
52                  if(outputAES[i][j] != ciphertext_check[i][j])
53                  {
54                    fault = 1;
55                  }
56                }
57              }
58
59              //determine the type of output
60              if(fault == 1 && detected == 1)
61              {
62                printf("Output error detected!\n");
63                num_wrong_detected++;
64              }
65              else if (fault == 1 && detected == 0)
66              {
67                printf("Output error not detected!\n");
68                num_wrong_undetected++;
69              }
70              else
71              {
72                printf("Output correct!\n");
73                num_correct++;
74              }
75            }
76          }
77        }
78      }
79    }
80
81    printf("Correct: %d\nIncorrect(detected): %d\nIncorrect(
      undetected): %d\n", num_correct, num_wrong_detected,
      num_wrong_undetected);
82  }
```

A.2 AES_Core.c

```
#include "AES_Core.h"

bool AES(uint8_t input[4][4], uint8_t outputAES[4][4], uint8_t
    outputCEC[4][4], int error_round, int error_col, int error_row
    , int error_bit, int error_bitdist)
{
  //setting buffers for AES
  uint8_t outputSB_AES[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t outputSR_AES[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t outputSR_AES_cor[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t outputMC_AES[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t outputAK_AES[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t outputAK_AES_cor[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};

  //setting buffers for CEC
  uint8_t input_enc[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t round_key[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t outputSB_CEC[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t outputSR_CEC[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t outputSR_CEC_cor[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t outputMC_CEC[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t outputAK_CEC[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  uint8_t outputAK_CEC_cor[4][4] =
    {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};

  bool reset = 1;
  bool detected = 0;

  //AES Initial step
  AddRoundKey(input, outputAK_AES, round_key);

  //CEC Initial step
  StateEncoder(input, input_enc);
  CECAddRoundKey(input_enc, outputAK_CEC, round_key);

  for(int i = 1; i <=10; i++)
  {
    //check if it's the first round
```

```
36      if(i != 1)
37      {
38        reset = 0;
39      }
40
41      //in the error round, insert a 2 bit error as specified in the
         input arguments
42      if(i == error_round)
43      {
44        bool bin_in[8] = {0,0,0,0,0,0,0,0};
45        HexToBin(outputAK_AES[error_col][error_row], bin_in);
46        bin_in[error_bit] = !bin_in[error_bit];
47        if((error_bit + error_bitdist) <= 7)
48        {
49          bin_in[error_bit + error_bitdist] = !bin_in[error_bit +
    error_bitdist];
50        }
51        else
52        {
53          bin_in[error_bit + error_bitdist - 8] = !bin_in[error_bit
    + error_bitdist - 8];
54        }
55
56        outputAK_AES[error_col][error_row] = BinToHex(bin_in);
57      }
58
59      //run the EDC for the input of SubBytes
60      detected = Correction(outputAK_AES, outputAK_CEC,
    outputAK_AES_cor, outputAK_CEC_cor, reset);
61
62      //run SubBytes
63      SubBytes(outputAK_AES_cor, outputSB_AES);
64      CECSubBytes(outputAK_CEC_cor, outputSB_CEC);
65
66      //run ShiftRows
67      ShiftRows(outputSB_AES, outputSR_AES);
68      CECShiftRows(outputSB_CEC, outputSR_CEC);
69
70      //run the EDC for the input of MixColumns
71      detected = Correction(outputSR_AES, outputSR_CEC,
    outputSR_AES_cor, outputSR_CEC_cor, 0);
72
73      //determine if it's the last round
74      if(i < 10)
75      {
76        //run MixColumns
77        MixColumns(outputSR_AES_cor, outputMC_AES);
78        CECMixColumns(outputSR_CEC_cor, outputMC_CEC);
79
80        //run AddRoundKey
81        AddRoundKey(outputMC_AES, outputAK_AES, round_key);
82        CECAddRoundKey(outputMC_CEC, outputAK_CEC, round_key);
83      }
84      else
```

```
85      {
86        //run AddRoundKey
87        AddRoundKey(outputSR_AES_cor, outputAES, round_key);
88        CECAddRoundKey(outputSR_CEC_cor, outputCEC, round_key);
89      }
90    }
91
92    //return 1 if there was an error detected, 0 if not
93    return detected;
94  }
```

B Timing Scheme ParTI

The table below is showing the timing scheme of the finite state machine and the round counter of the VHDL implementation of round-based AES protected by ParTI.

Time [ns]	State	Round Counter
10	S_INPUT	
20	S_INPUT2	
30	S_INIT	
40	S_WAIT	
50	S_ROUND	**1**
60	S_WAIT	
70	S_ROUND	**2**
80	S_WAIT	
90	S_ROUND	**3**
100	S_WAIT	
110	S_ROUND	**4**
120	S_WAIT	
130	S_ROUND	**5**
140	S_WAIT	
150	S_ROUND	**6**
160	S_WAIT	
170	S_ROUND	**7**
180	S_WAIT	
190	S_ROUND	**8**
200	S_WAIT	
210	S_LAST	**9**
220	S_OUTPUT	
230	S_OUTPUT2	**10**
240	S_DONE	DATA_READY
250	S_RESET	DATA_READY

Table B.1: Timing scheme of ParTI