A Cryptography Core Tolerant to DFA Fault Attacks

Carlos R. Moratelli[1], Érika Cota[1] and Marcelo S. Lubaszewski[2]

[1] Federal University of Rio Grande do Sul, Informatics Institute, Porto Alegre, RS, Brazil
e-mail: {crmoratelli}{erika}@inf.ufrgs.br
[2] Federal University of Rio Grande do Sul, Depto. of Electric Engineering, Porto Alegre, RS, Brazil.
e-mail: luba@ece.ufrgs.br

ABSTRACT

This work describes a hardware approach for the concurrent fault detection and error correction in a cryptographic core. It has been shown in the literature that transient faults injected in a cryptographic core can lead to the revelation of the encryption key using quite inexpensive equipments. This kind of attack is a real threat to tamper resistant devices like Smart Cards. To tackle such attacks, the cryptographic core must be immune to transient faults. In this work the DES algorithm is taken as a vulnerable cryptosystem case study. We show how an attack against DES is performed through a fault injection campaign. Then, a countermeasure based on partial hardware replication is proposed and applied to DES. Experimental results show the efficiency of the proposed scheme to protect DES against DFA fault attacks. Furthermore, the proposed solution is independent of implementation, and can be applied to other cryptographic algorithms, such as AES.

Index Terms: Smart Cards, Cryptography, Fault Attacks, Fault Tolerance.

1. INTRODUCTION

In many electronic systems, some type of information must be stored in the electronic device. For high security applications, encryption methods are usually implemented in the electronic device to protect the information that is stored in or transmitted to/from the device. Simpler devices, such as Smart Cards, implement the encryption mechanisms in hardware due to limitations in their embedded processors. However, the main cryptographic algorithms used nowadays present some weaknesses when implemented in hardware, thus being vulnerable to malicious attacks against the system.

Attacks against the cryptographic hardware can be divided into two classes: invasive and non-invasive. Invasive attacks are based on reverse engineering and require special laboratory equipments, making it too expensive. Non-invasive attacks, also called side channel attacks, exploit hardware implementation weaknesses [12]. One possible form of attack is to observe characteristics of the hardware during execution, such as power consumption, execution time or electromagnetic emissions, to extract statistical information that can eventually lead to the recovery of the secret key. For instance, Differential Power Analysis (DPA) exploits the power consumption of the device [8]. A second form of attack is called Differential Fault Analysis (DFA), or simply fault attack. This attack was proposed by Boneh, DeMillo and Lipton [3], and is based on the injection of a transient fault in the cryptosystem core. The attacker can use several techniques to inject transient faults in the cryptographic hardware. For instance, Glitch Attacks consist on submitting VCC (power), GND (ground) or the clock inputs to stress conditions. To do this, peaks on the power supply voltage or an irregular clock can be used. Another technique is the Light Attack. Intensive light (such as a laser, for example) may cause disturbances on semiconductors resulting in transient faults. Thus, fault attacks can be performed using very inexpensive equipment [14]. Different side channel attacks require different countermeasures. For example, preventing DPA attacks requires masking power consumption or modifying the algorithm to avoid correlation between consumption and input data. Solutions for DPA do not prevent DFA, timing analysis or electromagnetic attacks. Therefore, solutions for different attacks must coexist in the same cryptosystem.

Smart Cards are largely used in GSM mobile phones and are beginning to spread in banking and other applications. Such devices are an easy target to

DFA attacks for two reasons. They are implemented using high integration technologies, to make them portable and energy-efficient. Those technologies are becoming quite sensitive to transient faults [9]. Moreover, some of the most widely used cryptographic algorithms in Smart Cards are vulnerable to DFA attacks, including DES, AES and RSA [15]. Due to the vulnerabilities and the high demand for security of the applications that involve such devices, the study and the determination of real threads are extremely important. The exposure of possible vulnerabilities in the existent systems affects the credibility of such systems. Thus, the study of different kinds of attacks to the cryptographic hardware has highest importance, because the expansion and use of cryptographic devices depends highly on the approval and trust of the involved community.

In this work we propose a general and cost-effective solution to protect a hardware-based encryption system against DFA attacks (called simply fault attacks hereafter). To tackle such malicious attack, the cryptographic system needs to implement a hardware that is immune to transient faults. The proposed protection scheme uses partial hardware replication to annihilate the effects of transient faults. Experimental results presented for a DES cryptosystem show that protection is achieved at a low cost and without compromising the system performance. Furthermore, the proposed technique does not depend on the encryption algorithm and on its implementation.

The paper is organized as follows: Section 2 discusses related works. Section 3 shows how to break DES with a fault attack and presents some experimental results on that. Section 4 presents the proposed methodology to protect cryptographic cores and discusses its implementation and effectiveness in the case of DES. Finally, in Section 5 some conclusions are drawn and future works are discussed.

2. RELATED WORK

A few works have proposed techniques to protect cryptographic cores against fault attacks. Bertoni et al. propose in [1] the use of parity code to detect the injected transient fault on an AES core. For each AES operation on the input data, similar transformations on parity bits are performed. In addition, checkpoints are introduced in the algorithm to check the data integrity so that the encryption is suspended when a fault is detected. The area overhead is about 20% to protect the encryption module. The disadvantage of this technique is that system operation is suspended, which may not be desirable from the application point of view (if an attack is performed in the field or faults are caused by operation environment, for instance).

Breveglieri et. al in [4] combine parity code and hardware redundancy to detect and correct transient faults caused by either malicious attack or operation environment. Since the fault is corrected, system operation is not suspended. The overhead is about 40% for the fault detection circuitry and 134% for the entire fault detection and correction. However, the technique is directed to a special implementation of the AES algorithm. Moreover, that solution does not correct multiple faults. As we will show in Section 2.3, multiple faults are actually used to accelerate the fault attack and reduce the time necessary to reveal the key.

Thus, countermeasures capable of dealing with malicious fault attacks in cryptographic cores are still needed. Such techniques should be based on fault tolerance mechanisms resistant to multiple transient faults. Furthermore, they must avoid the suspension of the system operation, since system availability and dependability [6] are usual requirements of those applications. In our approach, the faults are detected and corrected in a way that is transparent for the user as well as the attacker. This provides a new level of security, where the attacker is unable to verify the efficiency of the fault injection procedure. Finally, it is important that the protection technique be independent from the algorithm implementation to ensure its applicability.

3. CRYPTOGRAPHIC ALGORITHM CODEBREAKING

Due to its simplicity, we use the Data Encryption Standard (DES) algorithm to explain how a fault attack is performed and how the system can be protected.

A. The DES Algorithm

The Data Encryption Standard (DES) is a symmetric key block cipher, and has a 64-bit input block, producing a 64-bit output cipher block. The secret key is 56 bits wide. DES uses two basic cryptographic techniques: *confusion* and *diffusion*. Confusion consists on replacing plain text blocks by other data blocks, while diffusion is simply a permutation of bits of the plain text. The algorithm is a combination of these two techniques applied to the plain text using the Key [13].

Figure 1 shows a block diagram of DES. Let us consider the processing of a single block (64 bits) from the initial plain text. The initial operation applied to the input is a permutation (*IP*) based on a 64-bit input table. This table is shown in [13]. Then, the result is split into two halves of 32 bits each (*L* and *R*). After this operation, encryption proceeds in 16 stages

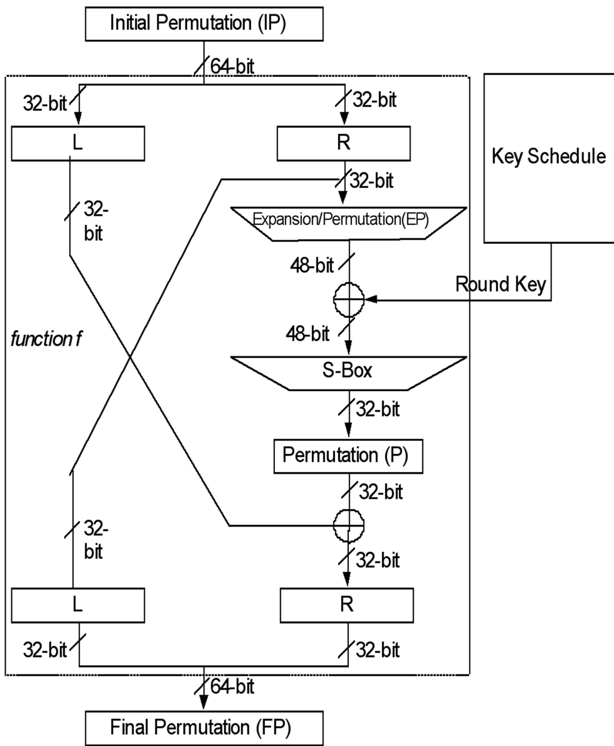

Figure 1. Block diagram of DES algorithm

or rounds, called *f function*. Each round updates the L and R registers to be used in the next round. In this process, the data block is combined with the key. Since the *f function* is the most important operation in the DES algorithm, all rounds should run and finish correctly. During each round, five operations are performed in the data block: expansion/permutation (EP), bitwise XOR with the key, substitution (S-Box), permutation (P) and, bitwise XOR with the left half of the input. After the last round, the left and right halves are exchanged, and, finally, the result is permuted in the final permutation (FP). Note that DES has only two registers (L and R). They are replicated in the bottom of Figure 1 to make it easier to understand. All other operations in the *f function* are implemented by combinational logic. Furthermore, each round uses a different subkey generated from the original 56-bit encryption key.

Inside the *f function*, the most important operation is the S-Box. This is a non-linear operation that provides security to DES. Figure 2 shows how S-Boxes work. The 48 bits resulting from the bitwise XOR involving the sub-key of that round are divided into eight 6-bit blocks and applied to 8 independent S-Boxes. Each S-box receives a 6-bit input and generates a 4-bit output. Input bits are substituted inside the S-Boxes using predefined rules. Every S-Box has its own substitution table as shown in [13]. Sixteen subkeys with 48 bits are generated, each one being used in a round of the *f function*. The algorithm for

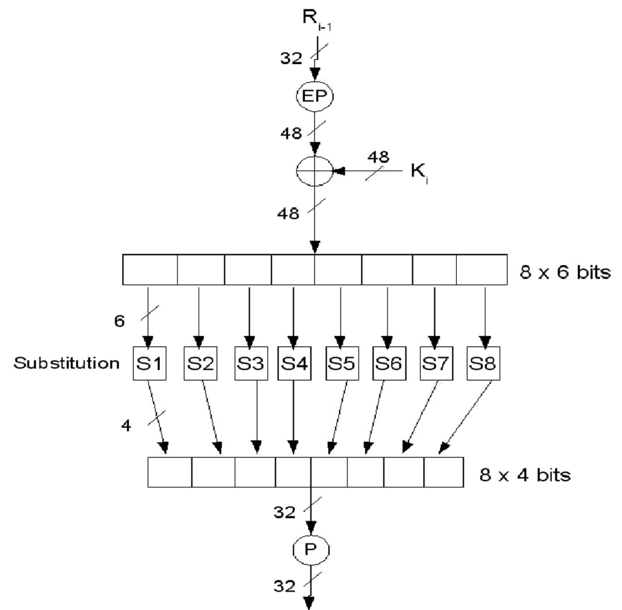

Figure 2. Details of DES S-Boxes

the generation of the subkeys is detailed in [13]. Decryption involves the same algorithm and the same original key, but subkeys are applied to the internal rounds in the reverse order [7].

B. Fault Attack on DES

This section shows how fault attacks are performed against DES. Initially, the attacker runs DES for a 64-bit block of a plain text and stores the resulting coded block. Then, DES is executed again for the same plain text while transient faults are injected in the hardware. Thus, the resulting coded blocks in DES output may be faulty. By comparing several faulty coded blocks against the correct coded block, the attacker can extract the secret key used inside the *f function* as will be shown next.

When a fault is injected in the R register in the 15th round of the DES execution, it affects specific bits of the coded output. Knowing the fault-free and the faulty coded words, and applying the algorithm, the attacker is able to identify some bits of the secret key. Repeatedly injecting faults in different bits of the R register and performing the analysis described above, the attacker will eventually expose all bits of the secret key. This process is detailed below.

Let us consider L_{15} and R_{15} the result of the 15th round of the *f function* in Figure 1. After executing the last round of the DES algorithm, L_{16} and R_{16} are given by the following equations:

$$R_{16} = P(S(EP(R_{15}) \oplus K_{16})) \oplus L_{15}$$
$$R_{15} = L_{16}$$
$$R_{16} = P(S(EP(L_{16}) \oplus K_{16})) \oplus L_{15} \quad (1)$$

where P is the permutation, S is the S-Box and EP is the expansion/permutation operation. In Equation 1, two variables are unknown: the subkey K_{16} and L_{15}. If a fault occurs in the right half of the dataflow (R register) during the 15th round, R_{15} will be replaced by a faulty R'_{15}, as follows:

$$R_{16} = P(S(EP(R'_{15}) \oplus K_{16})) \oplus L_{15}$$
$$R'_{15} = L'_{16}$$
$$R'_{16} = P(S(EP(L'_{16}) \oplus K_{16})) \oplus L_{15} \quad (2)$$

Equation 2 has also two unknown variables: the subkey K_{16} and L_{15}. Note that L_{16} and L'_{16} are the resulting coded word after the final permutation step of the algorithm, which can be easily performed backwards.

In order to find the value of K_{16}, one must eliminate L_{15}. This can be done by performing a bitwise XOR operation between Equations 1 and 2:

$$R_{16} \oplus R'_{16} = P(S(EP(L_{16}) \oplus K_{16})) \oplus L_{15} \oplus P(S(EP(L'_{16}) \oplus K_{16}) \oplus L_{15} = (S(EP(L_{16} \oplus K_{16})) \oplus P(S(EP(L'_{16}) \oplus K_{16})) \quad (3)$$

Thereby, the only unknown variable is K_{16}, which is a subkey generated from the sought secret key. The remaining variables are known outputs from DES. When K_{16} is known, it is possible to revert the subkey generation process and obtain 48 bits of the original key. The last 8 bits can be found by exhaustive search.

On the other hand, faults injected in any other round or any other part of the *f function* will not lead to the secret key in this fault model. More elaborated fault models can deal with faults in other rounds. Although the fault-free and the faulty coded words will be different after the 16th round, the application of the algorithm will not converge to the original encryption key. However, in a real attack, a fault injected into the system can hit any part of DES, at any iteration of the algorithm.

Thus, an attack consists in injecting single or multiple faults in the DES implementation and comparing the faulty output against the fault-free coded word. For each output, the algorithm presented in the next section is applied to reveal the subkey. The process is repeated until all 56 bits of the original key have been defined.

C. Fault Attack simulation on DES

To help in understanding how a fault attack can succeed, a Java-based software was implemented to accomplish the search of the subkey K_{16}. For the sake of simplicity, faults are injected in the right half of the data block (R register) and only in the 15th round. The attack starts by analyzing a cipher block without faults. Then it analyzes several erroneous cipher blocks by solving Equation 3.

A block diagram of the key search algorithm is shown in Figure 3. In the figure, the blocks that are inside the dotted line represent the dataflow for one algorithm iteration.

The algorithm execution starts by extracting R_{16} and L_{16} from the fault-free cipher text. These values are obtained by undoing the final permutation (*FP*), splitting the data block into two halves and exchanging the left half with the right half. These values are then used all along the software execution. The next step is to obtain R'_{16} and L'_{16} through a process similar to the one mentioned above.

Following the diagram in Figure 3, the *EP* operation is performed over L_{16} and L'_{16}. The result of the *EP* operation is a 48-bit block that must be XORed to the 48-bit round subkey K'_{16}, which is unknown. At this point, it is important to highlight the features of the DES S-Boxes, as mentioned previously in Section 3A The block resulting from the bitwise XOR with the round subkey is split into 8 blocks of 6 bits, since there are 8 independent S-Boxes. Due to the features of these S-Boxes, it is possible to divide the key search into 8 different search spaces. Thereby, the main goal of the key search algorithm is to test all 64 input possibilities for each S-Box.

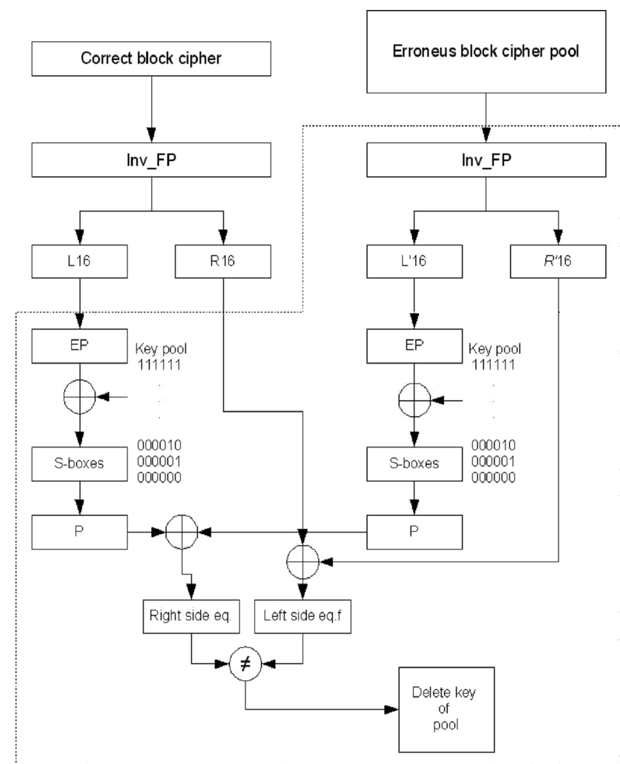

Figure 3. Block diagram of key search algorithm.

To test all these possibilities, one must firstly guess a 6-bit value for the key. Thus, the guessed 6-bit key is XORed to the 6-bit value resulting from the *EP* step and result is submitted to the respective S-Box. This procedure is performed for both cipher texts with and without errors, providing, at the end, two 4-bit blocks. Then, the *P* operation is performed for both blocks. Subsequently, a bitwise XOR is performed using these two blocks. The result is the right side of Equation 9. Similarly, a bitwise XOR is performed between R_{16} and R'_{16} resulting in the left side of Equation 9. Finally, the two sides of Equation 9 are compared. If they are equal, the 6-bit value guessed for the key is possibly the true key and should be tested with another cipher text block. If the result is not equal, the guessed key is not the correct one and can be discarded.

It can be noticed that a single fault affects at most two S-Boxes. For the unaffected S-Boxes, it is not possible to deduce any key value. Furthermore, the S-Boxes have 6 input and 4 output bits, and different input values can give the same output values. Then, for each S-Box affected by a fault, several key values exist that satisfy the equivalence given in Equation 3. Thus, one can understand why several erroneous cipher texts are needed to obtain the key K_{16}. For each erroneous cipher text analyzed, the key search space becomes smaller, since the keys that do not satisfy the equality are discarded. In the first iteration, the search space contains 8 x 2^6 possibilities. The algorithm then iteratively converges to a key that satisfies any input of Equation 3. This is the K_{16} key. Starting from K_{16} it is possible to obtain the 48 bits of the input key while the last 8 bits can be found by trying all remaining possibilities

During the simulation of fault attacks, faults were injected into the right half block during the 15th round, that is, R_{15}. Experiments were accomplished to determine the consequences of injecting a certain number of faults into R_{15}. These experiments consist in injecting from 1 to 8 faults in random positions of R_{15}. Our purpose is to analyze the amount of erroneous cipher blocks necessary to obtain the secret key. For each injection of a certain number of faults, 10 simulations were performed to determine the average number of erroneous cipher blocks necessary to obtain the secret key. The obtained results are presented in the graph shown in Figure 4. That plot basically shows that the more faults that are injected, the less erroneous cipher texts are needed. Starting from 6 simultaneous faults, the amount of needed blocks decreases slowly. This happens because faults affect bits of different S-Boxes for a same cipher block, thus revealing information about different parts of the key, and reducing the amount of cipher blocks needed. In an ideal fault injection campaign, each fault would affect a distinct bit of R_{15}. However, during a real attack, this highly accurate fault injection is improbable. Even so, by injecting single faults at random positions of R_{15} we have succeeded to obtain the secret key with less than 40 erroneous cipher blocks.

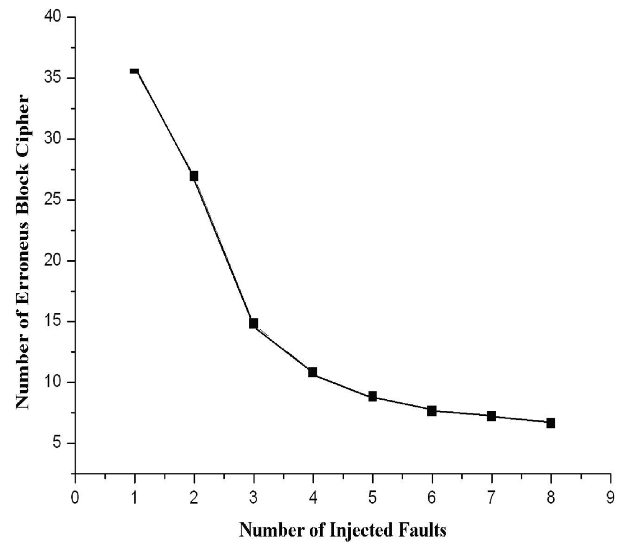

Figure 4. Relation between number of faults injected and number of cipher blocks needed to break the algorithm.

Cryptographic algorithms are developed so that a supposed attack based on the analysis of correct cipher blocks is computationally infeasible. However, previous experiments show that an attack can indeed be performed with very low computational cost against a hardware implementation of the DES algorithm. Therefore, we propose in the next section a protection scheme to neutralize the effects of a DFA attack to cryptographic cores.

4. CRYPTOGRAPHIC CORE PROTECTION

A. F function tolerant to fault attacks

Considering the simulation results obtained from fault attacks in the *f function* of DES, a countermeasure is proposed in this section to enhance the security of a cryptographic core implementing that algorithm. This approach is based on partial replication of the vulnerable parts of the core. Other possibilities could be total hardware duplication and time redundancy. The former has many drawbacks: First of all, the hardware area and power consumption are at least duplicated, which is not interesting (sometimes not even acceptable) in portable applications. Secondly, performance penalty is also a problem in the hardware duplication approach. The results are compared by a voter only in the end of the execution,

i.e., after the 16 rounds of the *f function*. Thus, if results of duplicated blocks are different, the whole execution must be repeated, increasing both execution time and energy consumption. Finally, the duplicated hardware is also vulnerable to fault attacks. A specialized attacker can attempt to inject transient faults into the two cryptographic processors simultaneously and may even obtain the secret key with less computational effort since the number of points for fault injection are largely increased and are strongly correlated in the two replicas.

To avoid partial or total hardware replication, it is possible to use a time redundancy approach that consists in computing the same input twice and comparing the resulting outputs. Nevertheless, Biham and Shamir [2] state that computing the encryption function twice is insufficient for high security systems, because the probability of the same fault to occur during both encryption processes may not be sufficiently low.

The countermeasure proposed in this paper consists in adding to the core a simple coprocessor that will assist the cryptography execution by detecting and correcting transient faults. A similar scheme was originally proposed by [5] to control the system frequency. In such work, the coprocessor assists the execution of the main processor by detecting and signaling fault occurrences. If the fault incidence is high, the coprocessor reduces de system frequency, aiming to reduce the number of faults. Thus, the system can operate at the highest possible frequency. In our proposal, we use the coprocessor to detect and correct the transient faults. Hereafter, the coprocessor will be called *assistant hardware*.

The assistant hardware must replicate only the most vulnerable parts of the encryption algorithm. For the DES algorithm, for instance, Section 3.2 has shown that the *f function* is the vulnerable part. So, the *f function* is replicated into the assistant hardware. The block diagram of the resulting system is shown in Figure 5. The arrows indicate system input and output signals.

In our context, as the *f function* is purely combinational, it is possible to implement the coprocessor as a combinational circuit. The assistant hardware consists basically in a replica of the *f function*, a comparator and a parity checker. The *f function* replica computes the same input data of the main processor at each round, aiming at assuring data integrity. The comparator is used to check the output data of the *f function* of the main processor against that of the assistant hardware at the end of each round. If the results are not equal, only that round operation is repeated. Finally, the parity checker verifies the integrity of the input data of the *f function* in the assistant hardware.

As shown in Figure 5, the assistant hardware performs the *f function* and compares its result against the corresponding result of the main processor. This comparison is purely combinational. Signals *data round* and *input key* are the input signals to the *f function*. At the end of each round, the results are compared by the assistant hardware using the *out data round* signal. This signal is the result of the *f function* of the main processor. If the results are not the same, the assistant hardware uses *elr* to signal to the main processor to perform the last round again. Before the execution of each round, the assistant hardware verifies the parity of the *data round* signal. If the parity is not the same, then a fault in the *f function* input occurred. The system cannot recover from this fault and (only in this case) execution is stopped. Parity error signals that the system must be restarted.

To support the proposed modifications, the main processor needs some adaptations. The main change to the original DES core is the addition of the *elr* signal. This signal indicates that the last *f function* round must be performed again. To do this, the main processor shall save the *f function* inputs in an auxiliary register. The parity signal is created to ensure the consistency of the data stored in this register. If a fault occurs in the *f function* inputs, the system cannot recover from it. Then, the execution is aborted. *Data round*, *out data round* and *out key round* signals correspond to data input of *f function*, data output, and cryptographic key, respectively, for each round. The other signals of the main processor are the original control signals of the DES core.

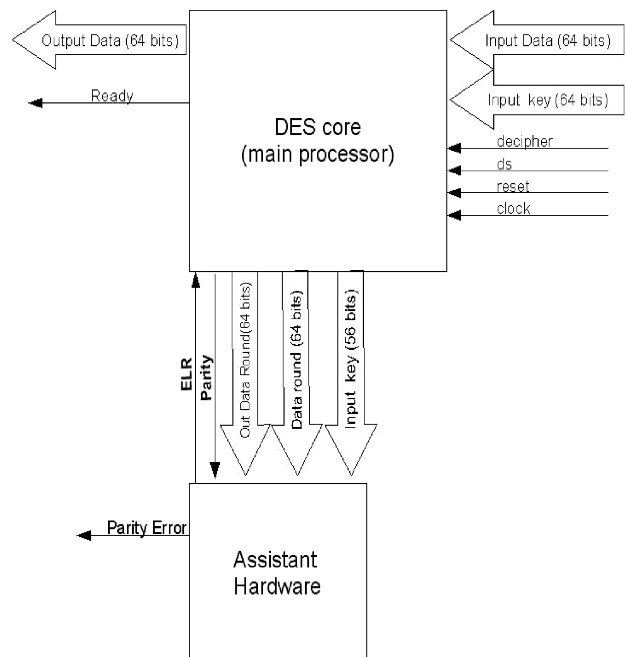

Figure 5. Block diagram of resulting system.

B. Experimental Results

An existing DES implementation obtained from [10] is re-used in this work. The main processor and the assistant hardware of the DES core were implemented in VHDL. Fault injection experiments using ModelSim XE III 6.0a were performed to check the level of security attained in the resulting cryptosystem. The VHDL description of the protected core is modified to include simple fault injection mechanisms based on duplicated registers and multiplexers. Faults are programmed into the additional registers whose contents replace, through the additional multiplexers, the contents of the original registers when the attack is simulated. This scheme allows the qualitative validation of the protected cryptosystem, helping to identify the protected and unprotected portions of the circuit and the level of protection achieved.

During simulation, faults were injected in registers *L*, *R*, their replicas, and in the inputs of the main processor. Table 1 summarizes our experimental results. Our fault injection campaign has demonstrated that the proposed solution allows the detection and correction of all transient faults inside the *f function*. These faults were injected in different locations and rounds of the cryptosystem execution. More specifically, faults were exhaustively injected in the *f function* registers in the 15th round, to ensure that malicious attacks are indeed avoided. In addition, faults in the inputs of the *f function* are also detected and, for those faults, execution is stopped. Faults occurring in a register of the datapath before or after the whole *f function* rounds are not detected and corrupt the output data. However, this corrupted output does not lead to the secret key. Thus, this protection scheme provides a fault coverage that is good enough to protect the *f function* against critical faults.

We note that the block that generates subkeys for each round of the *f function* is also vulnerable to fault attacks. However, the same fault analysis can be performed in this block to define the most critical parts to be duplicated. Some implementations of this core are built as a combinational circuit that generates every subkey during system initialization and stores them in registers. Those registers can be protected, for instance, by Hamming code and the assistant hardware composed of a Hamming checker. Other implementations generate a new subkey at each round. In this case, more detailed fault analysis must be performed.

Information about area and speed was obtained using the Leonardo synthesis tool and is presented in Table 2. One can observe a minimal performance penalty of 10% caused by the inclusion of the comparator at the end of *f function*, forcing the comparison to the assistant hardware in the same clock cycle. The area overhead of about 38% is quite small compared to the area overhead of a totally duplicated hardware.

Table 1: Behavior of protected cryptosystem in the occurrence of a fault attack.

Place of injected fault	Effect
Any register of datapath before *f function*	Undetected, the result is corrupted, but the secret key is not revealed.
In registers of *f function* inputs	Detected, execution is stopped, the result is not shown.
In signals inside *f function*	Detected and corrected.
In registers or signals of datapath after *f function*	Undetected, the result is corrupted, but the secret key is not revealed.
Signals of key generator core	Corrupts the result and can reveal the secret key.

Table 2: Comparison between the original core and the resulting system.

	Clock (Mhz)	Area (gates)
DES Core	165	3291
Resulting System	150	4561

We note also that the proposed method is not related to a specific implementation of the DES algorithm, since it is based on the logic instead of the VHDL code. Furthermore, it is possible to perform a fault attack analysis and define the most critical parts of other encryption algorithms such as AES to define its most vulnerable parts to be replicated. In [11], a number of possible attacks to AES algorithms are presented. In addition, fault injection tools considering permanent and transient faults have been largely studied in the last few years and can be reused and/or adapted for this analysis. Current work includes the use of partial replication as a protection scheme for the AES algorithm.

5. FINAL REMARKS

This paper has proposed a cost-effective solution to protect cryptographic cores against malicious attacks. First, a successful DFA attack was simulated in a Java-based platform, showing that hardware implementations of encryption algorithms have some weaknesses that can lead to the revelation of the secret encryption key. Then, a countermeasure has been proposed. It consists in replicating only the most critical parts of the logic. This scheme was implemented and validated for the DES core, resulting in 38% area overhead and 10% performance penalty. For this specific core, the implemented protection approach drastically reduced the probability of revealing the key by reducing the vulnerable area in the circuit. The main advantage of the proposed approach is that it is independ-

ent of the core implementation. Moreover, it can be easily applied to other algorithms, such as AES, which are also based on the iterative combination of the original data and the encryption key. However, only quantitative, randomized fault injection experiments can identify the statistical probability of the unprotected portions of the hardware to be attacked and reveal the cryptosystem secret key.

Current work includes the analysis and protection of the subkey generator core of the DES implementation and the application of the method to the AES algorithm. Furthermore, statistical fault analysis will be performed to quantitatively evaluate the protection of the encryption systems.

ACKNOWLEDGMENTS

This work has been partly supported by the Brazilian research agency CNPq. Thanks are also due to Dr. Luigi Carro for valuable discussions on the subject.

REFERENCES

[1] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri. A parity code based fault detection for an implementation of the advanced encryption standard. Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002., 2002.

[2] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. Lecture Notes in Computer Science, 1294:513{??, 1997.

[3] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. Lecture Notes in Computer Science, 1233:37{51, 1997.

[4] L. Breveglieri, I. Koren, and P. Maistri. Incorporating error detection and online reconfiguration into a regular architecture for the advanced encryption standard. Defect and Fault Tolerance in VLSI Systems. DFT 2005, 2005.

[5] T. A. Chris Weaver, Fadi Gebara and R. Brown. Remora: A dynamic self-tuning processor. University of Michigan CSE Technical Report CSE-TR-460-02, July 2002.

[6] J.-C. Laprie. Dependability of computer systems: concepts, limits, improvements. Software Reliability Engineering, 1995, pages 2{11, October 1995.

[7] A. J. Menezes. Handbook of Applied Cryptography. Boca Raton, 1997.

[8] D. Mesquista, J.-D. Techer, L. Torres, G. Sassatelli, G. Cambon, M. Robert, and F. Moraes. Current mask generation: A transistor level security against dpa attacks. 18th Symposium on Integrated Circuits and Systems Design (SBCCI2005), 2005.

[9] M. Nicolaidis. Design for soft-error mitigation. IEEE Transactions on Device and Materials Reliability, Sept 2002.

[10] Opencores. Opencores.org, 2005. Disponvel em: http://www.opencores.org. Acessado em Setembro de 2005.

[11] G. Piret and J.-J. Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. Cryptographic Hardware and Embedded Systems - CHES 2003, 2003.

[12] M. Renaudin and F. Bouesse. High security smartcards. Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04), 2004.

[13] B. Scheier. Applied Cryptography. John Wiley, 2rd edition, 1996.

[14] S. P. Skorobogatov and R. J. Anderson. Optical fault induction attacks. Cryptographic Hardware and Embedded Systems, 2002.

[15] W. Stallings. Cryptography and Network Security. Prentice Hall, 2rd edition, 1999.