Detection of hardware trojans and DFA attacks on cryptographic systems by applying residue checking

Ana Lasheras Mas

Date of defense: July 4th, 2019
Advisor: Ramon Canal Corretger
(Departament d'Arquitectura de Computadors)

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS
Specialization in High Performance Computing
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) – BarcelonaTech

Abstract

This thesis proposes the application of residue checking techniques on cryptographic accelerators to protect these circuits against malicious attacks. The focus is set in two different types of attacks. On one hand, the detection of hardware Trojans inserted during the design phase of a chip. On the other hand, finding errors injected during the execution of the system to avoid differential fault analysis attacks.

Hardware Trojans are malicious modifications of electronic circuits that can be introduced in any of their production phases and can affect the circuit in different ways. From all the different types of Trojans, we are addressing hardware Trojans introduced on the design or the fabrication phase of a chip focusing on the attacks where the variations affect the functionality of the circuit. For that reason, the evaluated systems are described in RTL using an HDL language. In this case, the proposed solution must be able to detect any change on the signals that the Trojan produces on the circuit and indicate somehow the system has been manipulated so it is not trustworthy.

Differential Fault Analysis (DFA) attacks try to obtain secret information from the execution of a cryptographic algorithm like other side channel attacks. The information is obtained by doing multiple executions for the same inputs where errors are injected during this execution. These executions generate a set of incorrect outputs through which sensitive information can be acquired. In this case, the proposal has to be able to perceive the alteration on the signals calculating the encrypted/decrypted value to repel the DFA attack by changing the output to a random value leaking no information to the attacker.

The technique used to detect these errors is called residue checking. Residue checking is a procedure initially defined to detect soft-errors in functional units. In this thesis, we propose to extend the usage of residue checking to security purposes and, specifically, to detect hardware Trojans and DFA attacks. To the best of our knowledge, this is the first time this combination is proposed.

In order to evaluate the proposal, three different cryptographic algorithms have been selected to apply residue checking (RSA, SHA and AES). These algorithms use distinct encryption schemes. We demonstrate how residue checking can be successfully applied in all of them with the aim of detecting hardware Trojans and repel DFA attacks.

First, we implemented the residue checker on the selected cryptographic circuits. Then, the protected circuit has been simulated to measure the error detection rate of the system with different configurations. Additionally, we have measured the overhead of the protection circuit in terms of operating frequency, power consumption and area. The goal of this assessment is to calculate how effective and costly is to apply the residue checking technique as a protection mechanism in cryptographic circuits.

Acknowledgements

First, I would like to thank my supervisor Ramon Canal Corretger for giving me the opportunity to work on this project, and also for all his advise during this master thesis.

I also would like to thank Luca Cassano from the Politecnico di Milano for helping us to contextualise the proposal of the project in the hardware security areas.

Furthermore, I thank my family for supporting me always, and for giving me the opportunity to do this master at UPC.

Contents

List of Figures

List of Tables

1. Introduction, motivation and goals

Nowadays, the number of computing devices is increasing and along with cyber-attacks are also rising [33]. These attacks can be set at different layers like in the application or operating system level, on the software layer, but also at the hardware level (Figure 1.1). One example of these attacks is the failure of a Syrian radar which microprocessors had a hidden *backdoor* inside that temporarily blocked the radar[1]. Another example of a hidden *backdoor* is the one in the chips used by Boeing 787 planes that could allow cyber-criminals to override and control the computer systems[4]. There are also examples of attacks on the field of security as the one on the random generator used for cryptography in Intel Ivy Bridge processors [36].

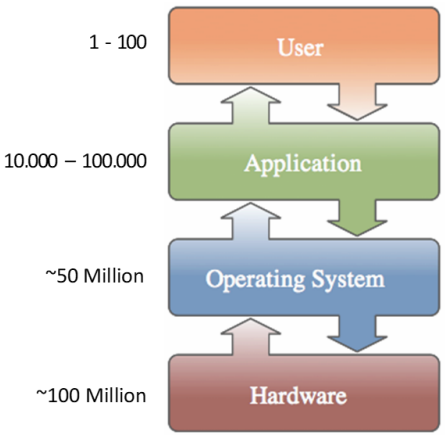


Figure 1.1: Estimated number of systems affected given a vulnerability per system level

Hardware has been always the target of reverse engineering for competitiveness between companies in the sector. However, in the last years, it has also become the target of malicious attacks in order to gain access to a computer or to extract information. Apart from the previous examples, there are much more critical vulnerabilities that have been discovered in modern processors like Spectre[29], Meltdown[32], RIDL[51] or Zombie Load[49]. These attacks can be prevented, once they had been discovered, through software patches. But, these patches are only a fix, not a solution. Using a patch can only be applied once the attack has been discovered implying that numerous system has been already compromised. Additionally, patches can degrade the performance of the system because to prevent the attack they have to change the initial behaviour of the system that it has been planned during the design phase. Therefore, new systems must tend to a secure by design approach where the security policies are embedded therein.

Hardware is the base of systems and all the layers built on top depends on it. For that reason, if the hardware is vulnerable, then the whole system is compromised that is why it should be protected.

Historically hardware has been a trusted source as all design and manufacturing were done by the same reliable company. Nevertheless, these days the high manufacturing costs have split the semiconductor industry into three kinds of companies: i) “Pure-play semiconductor foundries” which business is focus on the manufacturing process producing

integrated circuits (ICs) for other companies (e.g TSMC, GlobalFoundries), ii) "fabless" companies that do the design but do not produce the chip (e.g. Qualcomm, ARM) and iii) "integrated device manufacturers" that both design and manufacture (e.g. Intel, Samsung). This fragmentation implies multiple companies into the ICs production cycle(Figure 1.2) breaking the trust chain from the design to the product manufacturing. Consequently, security principles need to be put in place at the design time to anticipate vulnerabilities in order to guarantee a trustful product.

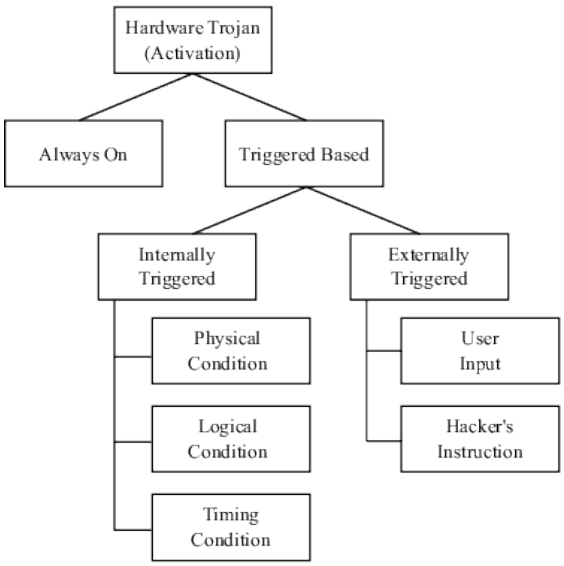
Figure 1.2: ICs production cycle

In the vulnerable process of chip production, many modifications can be inserted like hardware Trojans during the design or fabrication phase. After fabrication, fault injection can be used to obtain information from non-protected chips as in DFA attacks.

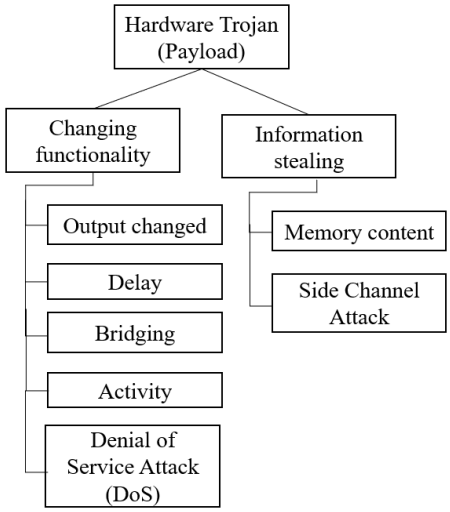
Two complementary approaches can be used to make this process more trustworthy: i) white box approach when the design is completely made by the trusted company and ii) black box approach applied when the product contains multiple modules (IPs) designed by others. The white box approach is based on inserting the corresponding security functionalities during the design phase to guarantee the service once manufactured elsewhere. The black box approach extends the design based on third-party IPs with extra functionalities to ensure the trust in the non-trusted parts.

1.1 Hardware Trojans

Hardware Trojans are malicious circuits place on a trusted chip that can cause unexpected behaviours on the regular operation of the system.



(a) Trigger[30]



(b) Payload

Figure 1.3: Classification of hardware Trojans

Hardware Trojans can be classified based on how they are triggered and their payload[10] (Figure 1.3). This malicious circuits can be triggered after i) a combination of conditions what is also called logical condition triggered or ii) a sequence of events during execution(Figure 1.3a). In case the Trojan is activated after a sequence of events, these occurrences can be based on i) a physical condition like the input of the user or a change on the activity of the circuit like rising the temperature of the system; or ii) time-based .

Based on how Trojans affect the execution of the system (Figure 1.3b), they can perform two different actions: i) change the intended functionality or ii) steal information . Changing the functionality of the circuit includes: i) returning a non-desire output, ii) including a delay on the system, iii) a bridging fault that connects two signals of a system to output the dominant signal, iv) changing the operational activity like consuming more power what could drain the battery of the device or v) denying the service (DoS) where the chip will stop running . Information-stealing attacks can be applied using a side channel attack to get through this external circuit relevant information without notifying the user or by reading *sensitive* content from memory.

Figure 1.4 shows an example of a hardware Trojan characterise as a combinational triggered - after a combination on the inputs A and B that change the original output C.

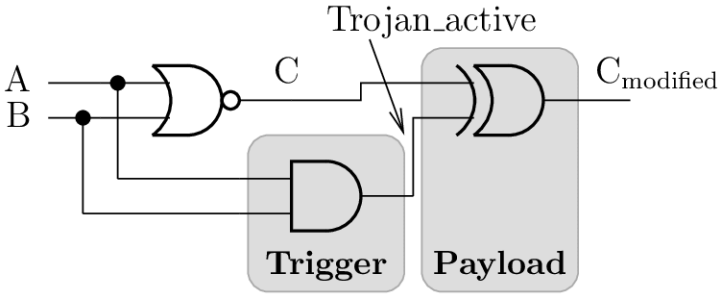


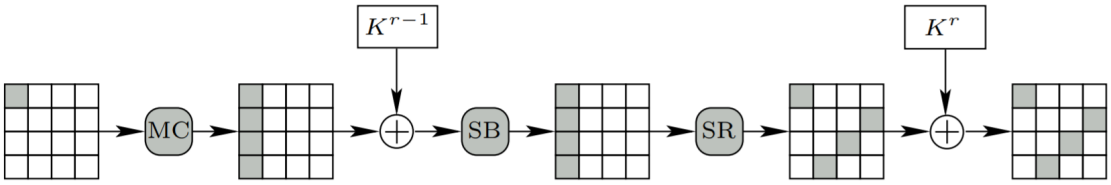
Figure 1.4: Example of a functional hardware Trojan[9]

Hardware Trojans are usually difficult to detect as they hide in the circuit, they trigger on a specific unknown condition and they can affect the circuit in different ways. Moreover, as devices are becoming smaller and more complex, it is difficult to detect them. Additionally, millions of chips are fabricated and it is not possible to test all these chips one by one to ensure they do not include any hardware Trojan.

1.2 Differential Fault Analysis (DFA)

DFA attacks are applied to cryptography circuits as a type of side channel attacks to steal sensitive information such as the encryption key [27]. DFA is based on injecting maliciously erroneous values into a cryptographic circuit while it is in execution multiple times and then collecting all these incorrect outputs to analyse them and infer secret information like the internal state of the system or cryptographic keys. This attack is denoted as a white-box attack because the attacker has access to the target code and it can manipulate it to interfere on the execution using developer tools such as debuggers or emulators[56].

An example of how DFA works on the Advanced Encryption System (AES) algorithm is shown in figure 1.5. The attacker injects a fault in a certain bit and knowing which operations are done to calculate the output of the cryptographic algorithm it can determine which bits will change from the original output as it shows subfigure 1.5a. Additionally, from the output with faults, the correct output and knowing where the errors were injected, along with an analysis of the algorithm behaviour it can be determined some certain subkeys of the algorithm that will lead to discovering the cipher key (Subfigure 1.5b).



(a) Analysis of how the output changes based on the input in AES[15]

Output with faults					Output without fault					Error			
DE	02	DC	19	$\oplus$	39	02	DC	19	=	E7	00	00	00
25	DC	11	3B		25	DC	11	6A		00	00	00	51
84	09	C2	0B		84	09	85	0B		00	00	47	00
1D	62	97	32		1D	FB	97	32		00	99	00	00

(b) Erroneous output based on a fault injection [13]

Figure 1.5: Example of a Differential Fault Analysis attack on AES

DFA has been demonstrated to be a very effective attack technique as it allows the attacker to choose which faults to inject and at which place in the circuit to insert them. Also, because it can attack several cryptographic algorithms like the symmetric key algorithms AES[15] or RSA[11]. The attacker just needs to know how the cryptography algorithm behaves to calculate the amount of insertion to do and the analysis to be done from the collected data to get the secret key. But, even if the cryptography algorithm is not known by the attacker doing multiple executions with different injections can help the attacker by analysing the output of how the algorithm behaves and steal secret information based on that[11].

1.3 Objectives of this thesis

The main objective of this thesis is to implement a method at the HDL level to detect hardware Trojans changing the system functionality and DFA attacks on cryptographic algorithms. The implemented residue checker should be able to detect security attacks extending the application of this technique to security from the original soft-error detection [37]. This method should be as general as possible to be applied to multiple types of cryptographic algorithms such as symmetric-key algorithm and hash functions.

Another goal of this project is to measure the coverage, cost, and impact of the method proposed. Consequently, it will be evaluated the overhead in area, energy consumption and performance impact on an FPGA substrate.

2. State of the art

This chapter shows an overview of the recent research done to detect hardware attacks. In concrete, it focuses on the two attacks this thesis faces that are hardware Trojans and DFA. There is also a synopsis about the work done related to residue checking and the fields in which residue checking was already applied.

This chapter concludes with the progress beyond the state of the art of this thesis.

2.1 Techniques to detect hardware Trojans

Over the last years, several kinds of techniques have been presented that detect a certain type of hardware Trojans. These proposals can be classified in four different techniques to detect Trojans: i) side channel analysis, ii) logic testing iii) machine learning and iv) formal methods . However, there are other proposals on the literature such as runtime methods, reverse engineering and Trojan activation methods[7].

Side channel analysis techniques measure non-functional properties of the manufactured chip in operation such as power consumption, leakage, temperature, delay or electromagnetic radiation and compare these values with the ones from the trusted design or manufactured chip. This technique is not good for small circuits as it is susceptible to experimental and process variation noise.

Logic testing is based on triggering a rare event on various internal nodes and propagate it to an output node to observe its effect. This method is not affected by process variation and noise effects but it has the difficulty to define correctly the test vector for that reason is a good option for small circuits.

Machine learning techniques study elements on the circuit such as netlists in order to detect Trojan features and then perform the detection task effectively based on the learned patterns during the circuit analysis phase.

Formal methods verify the design of a circuit to fulfil all the requirements specified. These methods are applied to the pre-synthesised hardware designs.

2.1.1 Side channel analysis techniques

Atieh Amelian and Shahram Etemadi Borujeni[3] present a method based on side-channel analysis to detect hardware Trojans measuring path delay. The method does not modify the target circuit and it can be applied to test the IC after its manufacturing. The method calculates the delay on a set of selected paths of the circuit and then comparing the expected delay of the Trojan-free design with the delay on the same paths of the manufactured IC it can decide if any hardware Trojan has been inserted during fabrication. This method detects more than 80% of Trojans using a configuration of 20 paths with an accuracy of 0.01 ns.

Nguyen, Luong N., et al.[38] propose to use backscattering (i.e. the reflection of signals back to their original direction) for hardware Trojan detection as this signal can provide information about the ongoing state of a system. The goal of the paper is to can detect dormant Trojans (i.e not active Trojans) on circuits to determine possible malicious modifications on ICs without needing to identify the activation condition from the many possible triggers. The proposal is evaluated along with cross-training using the Trojan-free design three different Trojans from TrustHub[41] that leak information or do a DoS attack. Evaluation results show that all three cases have been correctly detected the Trojan.

Hossain, Fakir Sharif, et al. [22] specify three detection methods based on a side-channel analysing power consumption. One of the methods is a modification of scan segmentation methodology[23], previously proposed by the authors. Another method is Equal-Power Self-referencing (EP) that obtains gating points with EP from the design and then it applies the detected EP patterns and measures the generated currents individually. The third method proposed is Equal-Power Neighbouring self-referencing (EPN) and it tries to improve detection sensitivity on circuits with many process variations. The system is evaluated by inserting two different Trojans from TrustHub[41] into an AES without altering the original layout of the circuit. The results obtained differ from one circuit and another giving better results the EPN algorithm in all the tested cases.

2.1.2 Logic testing techniques

Amin Bazzazi et al.[7] define a way to localise hardware Trojans based on the logical values of nodes at runtime combining logic testing and runtime techniques for hardware Trojan detection. The method analyses the circuit and obtains a set of nodes with the greatest similarity on their logical values. Then, it compares the values of these nodes with the ones from the known non-manipulate circuit to detect if the evaluated circuit includes hardware Trojans. The proposal detects more than 80% of the Trojans on the evaluated circuits increasing the power consumption a 13% and the area a 15% from the initial value.

Huang, Yuanwen, Swarup Bhunia, and Prabhat Mishra[24] propose a method to generate efficient test vectors to detect hardware Trojans using side-channel analysis. The generated test vectors maximise the switching activity of the inserted Trojans while it minimises the switching activity in the rest of the circuit to make easy hardware Trojan detection applying side-channel analysis. Their proposals improve the Trojans detection with side channel analysis by more than 96%.

Salmani, Hassan, Mohammad Tehranipoor, and Jim Plusquellic[48] present a technique to raise the hardware Trojan activity inserted on a circuit and, consequently, decrease the activation time of these Trojans facilitating its detection. Hardware Trojan detection is improved based on an analysis of the circuit modelling the transitions on the system and the number of clock cycles with a geometric distribution. Additionally, they prove that the insertion of dummy flip-flops on the circuit can help on increasing transitions what also grows Trojan activation.

2.1.3 Detection applying machine learning techniques

Hasegawa, Kento, Masao Yanagisawa, and Nozomu Togawa[20] describe a machine learning based hardware Trojan classifier detecting Trojans nets inserted at gate-level during in ICs design. The system is tested on hardware Trojans from TrustHub[41]. All the Trojans evaluated produced a denial of service attack and are triggered based on timing or physical conditions. The evaluation shows that they are able to determine Trojan nets with 92.2% of precision and 74.6% of F-measure.

Yoon, Junghwan, et al.[58] introduce a technique to detect hardware Trojans in FPGAs doing a statistical analysis by applying machine learning. The analysis is done in a forward and backward direction. Forward analysis directly analyses the netlists at a synthesis phase while backward analysis checks the circuit reversely from bit streams. Multiple aspects are proposed to be considered in the synthesised design to detect hardware Trojans such as vectorized boolean equation of the LookUp Tables (LUTs), number of LUT's input pins, type of LUT's output pins, number of LUTs and nets from input and output pin to the Input/Output Blocks (IOB), number of LUT that affect the output. This technique is described to be promising based on other results applying their detection tool but there are no results given regarding hardware Trojan detection.

2.1.4 Detection applying formal verification methods

J. Rajendran, A. Dhandayuthapany, V. Vedula and R. Karri[45] proposes a formal verification method to detect hardware Trojans inserted on third-party IPs and its trigger condition. This technique can detect hardware Trojans leaking sensitive information from the circuit by applying model checking techniques. In order to apply its proposal needs to have a temporal logic description of the design. This mechanism detects Trojans as long as the leakage of information is inside the number of clock cycles analysed.

Xiaolong Guo, Huifeng Zhu, Yier Jin and Xuan Zhang[18] describe a technique that it formally verifies a circuit to expose leakage paths at the circuit level. This technique uses information flow tracking methods to understand how the information in the circuit flows and allowing to successfully detect Trojans or other vulnerabilities in the system.

Jonathan Cruz, Farimah Farahmandi, Alif Ahmed and Prabhat Mishra[12] explains a technique that detects hardware Trojans by analysing the design based on combining automatic test pattern generation method and model checking.

2.2 Techniques to protect against DFA attacks

DFA attacks seem to be a very effective attack against ciphers. In the literature, several countermeasures have been proposed based on redundancy. These proposals can be categorised based on the where they are implemented that it can be i) a dedicated device detecting or blocking the attack, ii) modifications on the circuit computation, and iii) protocol level solutions that defining new algorithms in which there is a low probability of fault .

This section focuses on the last proposals shown in academia to protect circuits against DFA attacks.

2.2.1 Protection based on external devices

He, Wei, Jakub Breier, and Shivam Bhasin describe a detection logic to protect systems against fault injection attacks using a laser to inject the faults. The proposal uses a high-frequency digital Ring-Oscillator watchdog and a disturbance sensor. Their results indicate that 94.20% of fault attacks tested on ciphers can be detected[21].

2.2.2 Protection based on code modifications

Indrani Roy et al. define a framework to detect sensitive points of a block cipher design and fix these points based on the cipher specification and a set of security requirements. The implementation is based on the fact that cryptographic algorithms have different vulnerability depending on the point the faults are inserted and trying to protect these more critical locations. The evaluations is done on three cryptographic algorithms i) AES, ii) CAMELLIA and iii) CLEFIA giving a fault coverage between 50% and 100% depending on the online security margin chosen[47].

2.2.3 Protection based on new protocols

Christof Beierle et al. present a new lightweight cryptography algorithm secured against DFA attacks. Lightweight ciphers pretend to provide efficient cryptography algorithms with some of these algorithms are based on well-known symmetric algorithms. In concrete, this proposal follows the structure of AES to define its encryption and decryption scheme[8].

Baksi, Anubhab, et al. propose a protocol strategy that ensures faults are inserted into some part of the block cipher plain text instead of an intermedium cipher value increasing the difficulty to obtain information from a DFA attack. Additionally, this technique can be easily applied to standard ciphers such as AES[6].

2.3 Residue checking

Residue checking is based on the mathematical properties of integer modular arithmetic. Addition, subtraction and multiplication operations with integers follow this property (2.1).

$$((a + b)\ mod\ N)\ \equiv\ (a\ mod\ N) + (b\ mod\ N) \tag{2.1}$$

This concept was originally applied to adder circuits[43] by W. W. Peterson in 1957 with the objective of detecting failures on components. Residue checker components allow to protect arithmetic circuits using a smaller quantity of additional hardware as the operations can be replicated with a smaller amount of bits than the original operation but getting the same protection results thanks to this property.

Since then, many other papers have been written following the original idea of protecting arithmetic modules such as adders or ALUs[39] but there are also other proposals focusing on other areas like soft-error detection[40].

2.3.1 Application of residue checker in the last years

Daniel Lipetz and Eric Schwarz propose the use of residue checker for decimal floating-point units protecting the two values representing the float one in base 10 and the other in base 2. They also give a discussion on the coverage obtain depending on the modules selected to protect the circuit[31].

Shugang Wei presents an improve version of residue checker for Multiply–accumulate operations with signed-digits. The calculation of the correction with residue checker is used using a tree structure. No concrete values on the coverage are given but they analyse a set of configurations indicate the best one based on coverage and area utilisation [53].

Stanisław J. Piestrak and Piotr Patronik explain their proposal to protect against faults digital finite input response (FIR) filters. FIRs are a kind of filters use for Digital Signal Processing. Their implementations show a 100% coverage of errors with only a 3% increase on area[44].

2.4 Novelty of this thesis

This thesis proposes a novel method to detect attacks on cryptographic systems. As far as we know, residue checking has never been used in the area of security to detect any kind of attack including the two attacks this thesis focus on that are hardware Trojan and DFA attacks.

Hardware Trojans are still an issue on real circuits as subsection 2.1 has detailed. Most of these articles based on the detection of any kind of hardware Trojans such as information stealing which is an objective of this thesis. The proposed detection technique varies from invasive and non-invasive methods so our proposal based on an invasive method is a valid approach at the moment. Even though the main objective in security area is protected, most of this methods not only centre its attention on giving a good detection result but also to minimise the area overhead. For that reason, our goal is to implement an attacks detection mechanism which overhead is as minimum as possible.

DFA attacks are one of the main threats to cryptographic circuits these days. Section 2.2 presents various techniques to detect them and protect the circuits against them, but there are recent papers describing new effective and efficient DFA attacks. Consequently, it is of utmost importance to contribute with new detection mechanism to this type of attack. Most of the papers reviewed in section 2.2 focus on AES unpatent cryptographic algorithm and other proprietary ciphers such as CAMELLIA from Mitsubishi electrics and NTT; or CLEFIA from Sony. However, there are many other standard cryptographic algorithms such as RSA and SHA mostly used nowadays[16] but which protection has not been investigated as much in the last years.

In conclusion, hardware Trojans and DFA attacks are still a threat to ICs so new techniques must be proposed to protect these systems against them. For that reason, the proposal of a novel method to detect them as this thesis can contribute to reduce the impact of hardware attacks on the security area.

3. Methodology

This chapter defines the method used to implement and evaluate the proposal of this thesis including how the target attacks were chosen, the tools used and the steps followed on the implementation and evaluation. It also includes theoretical values of the level of protection that the approach should give based on the attack and why.

3.1 Implementation

3.1.1 Initial decisions and stages of implementation

The first step on the implementation was to decide the attacks to target. We defined the initial target of this thesis was to detect hardware Trojans which payload changes somehow the functionality of the system. And, after that, it was extended to also detect other attacks affecting the values on the signals of the circuit, in concrete, DFA attacks.

Along with the attacks to detect, the cryptographic algorithm to protect were also selected. This decision was made based on the benchmarks provided by TrustHub[41] that had already some algorithms with Trojans. TrustHub provides benchmarks for two cryptographic algorithms AES and RSA. So, these two algorithms were chosen to protect them and also the hash function SHA-1 as we had already a design of a cryptographic accelerator implementing this algorithm and even though this algorithm is not secure anymore[5] it was chosen as it was still supported for digital signatures[2, 35] and also as its operation are quite similar to other versions of SHA so it can be used to probe the concept.

Once the target attacks and cryptographic algorithm were chosen, the residue checker mechanism and other modules needed were implemented and tested for the algorithms. Residue checker was not done straight forward as not all the algorithms use arithmetic operations supporting modular arithmetic but bitwise operations that do not follow these properties so some modifications to our residue checker were applied. More details on how the residue checker mechanism has been adapted to the algorithms, the defined modules and its insertion in the ciphers original design are described in the following section 4.

3.1.2 Development tools

The only tool used for the implementation was an Integrated Development Environment (IDE) from Xilinx called Vivado Design Suite[57]. This decision was taken from the many possible IDEs supporting HDL because we already know how to use it from previous projects and work. Another point why it was chosen is because it is a very complete tool that allows coding through the interface checking the syntax errors and simulation to can corroborate that the design is correct.

3.2 Evaluation

3.2.1 Initial decisions and stages of evaluation

The first step on the evaluation once the implementation was finished was to verify the system under hardware Trojans and DFA attacks.

Hardware Trojans to detect were taken from a set of TrustHub's benchmarks previously modified to include our residue checker. Our residue checker was not applied to all the available benchmarks as it has a limitation that it can only detect hardware Trojans that change the functionality. So, side channel attack Trojans could not be detected.

The detection of DFA attacks was done by simulating the detection of changes on the registers of the design. This was done by defining a testbench for each one of the evaluated systems and performing the simulations of them.

Then, other metrics were measure, in concrete, area, energy consumption, and the maximum frequency of operation. All these metrics were measured after performing the synthesis of the design on a certain FPGA model. Another measure done was to check if the integration of residue checker on the circuit affects the critical path of the original system to see if the performance, in terms of time, is degraded.

Evaluation details about how the simulations and the metrics have been executed are described on section 5 along with the results obtained for each one of the assess metrics.

3.2.2 Evaluation tools

The tool used for the evaluation was the same one as for the implementation, Vivado Design Suite [57]. This tool includes a simulator, and it also allows to synthesise and implements a design on a certain FPGA. Additionally, it provides automatic metric's calculation of the design after its synthesis or implementation such as power consumption, area utilisation and timing constraints. One disadvantage of this tool is that the software is proprietary so some FPGAs and options are only available using the correspondent unlimited license or using a free trial license that gives you 30 days of free use.

4. Development of the proposal

This chapter explains in detail all the steps involved in the implementation of the proposal that was summarised on the previous section 3.1. These steps include how the modules for detection and insertion of faults has been implemented, the constraints and limitations that the proposal has, how these limitations affect the system, and how the proposal has been specifically applied to each of the cryptographic algorithms.

4.1 Analysis and re-organisation of the cryptographic cores

Once the cryptographic algorithms were selected, they had to be analysed to learn how each one of the algorithms behaves and which is the purpose of the different signals in the design to define which ones can be protected and how to apply the residue checker mechanism to them.

For each one of the algorithm, all the signals were classified between control signals and operational registers. Control signals indicate to the system how to perform and are usually composed of a small number of bits, mainly only one bit. For that reason, control signals were not our focus of protection as they had to be completely replicated and also because there are usually not a focus of attacks as they are commonly implemented as wires, not on storage element where faults can be captured. Operational registers are used to keep the partial calculation of the cipher algorithm. The value of registers is usually a set of bits, in the cases of the algorithms selected, most of the registers have a size of 32-bits or more. Additionally, faults can be visible and propagated through them over the calculations. Consequently, all the registers from the algorithm are a possible candidate for protection. Ideally, all registers of the systems should be protected but there were some issues, that is explained in subsection 4.1.1, forbidding.

After the analysis, we inserted the extra modules needed to implement and evaluate the fault-detection mechanism. The two modules inserted are the proposed version of the residue checker mechanism to protect the circuit, and a fault-injector module to can verify the correct operation of the fault-detection mechanism that it is our implementation of the residue checker (Figure 4.1). Both modules are explained in detail on the following subsections 4.1.1 and 4.1.2.

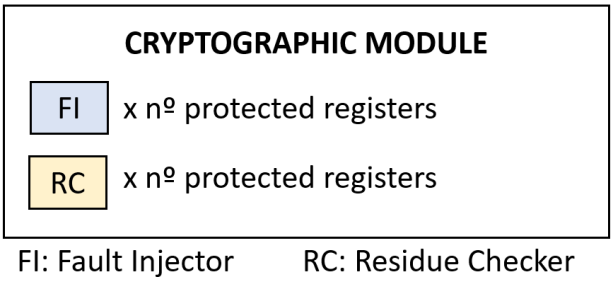


Figure 4.1: Modules inserted within the cryptographic module design

4.1.1 Residue Checking and Rotary Residue Checking

Residue checking is applied to each one of the operations with registers that can be protected. This technique is applied in two phases: i) protection and ii) check . The protection part is base on defining additional registers with the modulo value of each one of the registers, calculating in parallel the value of the original register and the modulo registers through the algorithm. The check part is based on checking the equality of modules (2.1). If there is an inequality detected this means that there could be a fault injected on that operation (Figure 4.2).

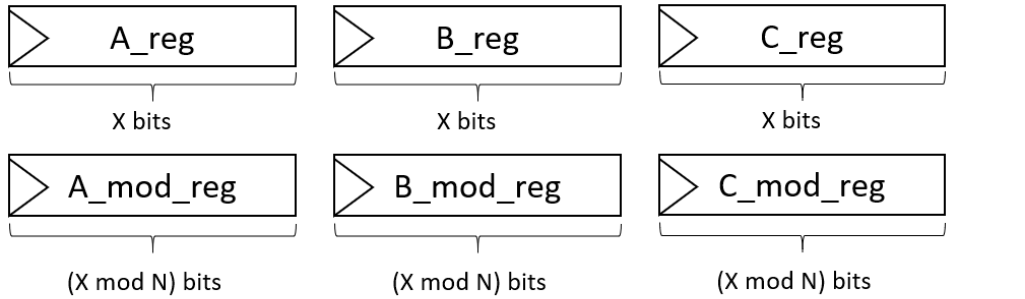


Figure 4.2: Explanation of the residue checker mechanism

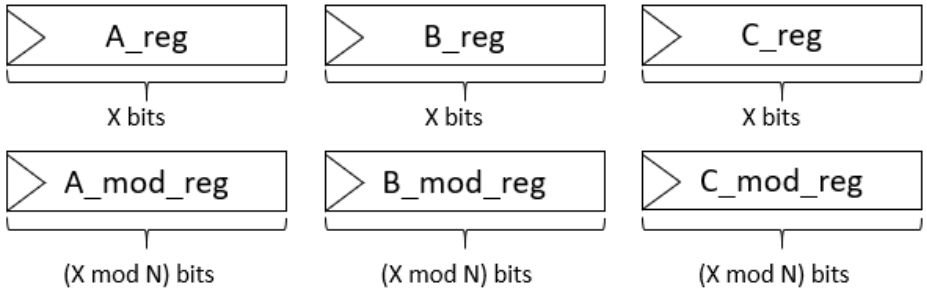
The explained residue checker is based on a property that can only be applied to certain arithmetic operations. For that reason, the operation types used in each one of the algorithms have to be taken into account because depending on the kind of operation residue checker cannot be directly applied as not all operations follow the same properties.

Two of the chosen algorithms, SHA-1 and AES, use bitwise operations and permutations while RSA is mostly based on additions and subtractions. Bitwise operations and permutations change the output in a way that the modular arithmetic property use in residue checker is not fulfilled. For that reason, it is not possible to apply straightforward the residue checker concept to those two algorithms. Therefore, residue checker for these SHA-1 and AES can only be applied to bitwise operations (4.1), not to permutations, with the limitation of only using power-of-two values for the module depicted as N in (4.1).

$$(a\ AND\ b)\ mod\ N \equiv (c\ mod\ N) \equiv (a\ mod\ N)\ AND\ (b\ mod\ N)\ forall\ N\ in\ 2^n \quad (4.1)$$

This limitation on residue checker reduces its performance as the use of power-of-two values for the module is like just protecting the N last bits of each register leaving the other most significant bits unprotected. This does not happen using odd values for the module as the remainder is calculated based on all the bits of the register. As a result of this limitation, an alternative version of residue checker has been proposed for these cases that we have called the rotary residue checker.

The Rotary residue checker is based on the same principle of the residue checker applied to power-of-two modules but including a rotation mechanism to change the bit that will be protected after a predefined number of cycles. Figure 4.3 shows visually how the rotary residue checker behaves by protecting a different set of bits depending on the execution cycle. The picture shows the simplest of the condition to update the bits to protect that is just adding N to the initial bit index and resetting these values when needed the more significant bits has already protected. This condition fits if the module value, N, is multiple of the total number of bits that the original register has, X. However, in case N is (X/2) + 1, one bit more than the original size of the register then the most significant will be always unprotected. This case has been taken into account on the implemented rotary residue checker, but it is not depicted to simplify the pseudo-code of the picture.



Protection phase: Calculate operations based on the module

```
A = B OR C
A_mod = B_mod OR C_mod

If  (timeToChangeBits == true)
    A_mod = A((iniBit + N - 1)..iniBit)
    B_mod = B((iniBit + N - 1)..iniBit)
    C_mod = C((iniBit + N - 1)..iniBit)
    If ((iniBit + N) < X) → iniBit = iniBit + N
    Else → iniBit = 0
```

Check phase: Corroborate that the operations are not attacked

```
If (A((iniBit + N - 1)..iniBit) != A_mod) → ERROR !
```

Figure 4.3: Explanation of the rotary residue checker mechanism

4.1.2 Fault injector

The fault injection module is designed to return an output value based on its input with a certain bit change on it. In concrete, the fault injection component defined is based on the following specification (Figure 4.4).

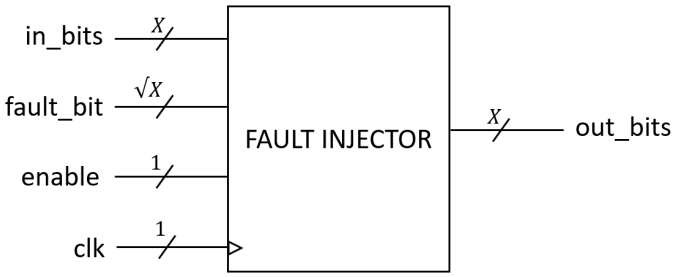


Figure 4.4: RTL schematic of the fault injector component

The control input signals are the clock to update the output only at the rising edge of the clock, and an enable signal that allows activating the fault injector or just let the input go by. The other two inputs are the input value to be attacked and an input position to know where to insert the fault. The output value of the module behaves differently based on the enable signal. If the enable signal is activated, the output is the input value with the bit on the indicated position flipped. In case it is deactivated, the output is the input value with no modifications. Figure 4.5 shows a simplified internal representation of the fault injector in case the input signals has a size of 2 bits.

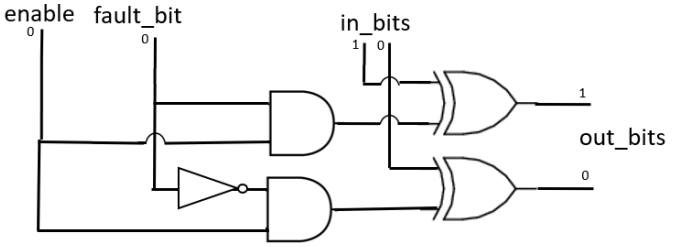


Figure 4.5: Internal design based on logic gates of the fault injector component for an input of size 2 bits

This component is used to test that the fault-detection mechanism behaves as it was defined. This fault injector is instantiated multiple times on each design, specifically, there is one instance one for each of the values protected.

The injector has been designed on a parameterized way to enable its use for different input sizes. Additionally, it is as simple as possible and it can easily be extended or changed. For this reason, it only flips one of the bits in the input signal and it inserts the fault when it enables instead of having an extra signal indicating the time. This control of the time when the fault is inserted is done outside this component and it is further explained in the evaluation section 5.

4.2 Protection of RSA applying residue checking

4.2.1 Algorithm

RSA is a cryptographic algorithm named after its designers Rivest, Shamir and Adleman. This cryptosystem was originally presented as one of the first asymmetric algorithms for cryptography in 1977[46]. Unlike symmetric cryptography algorithms, where the same secret key is used for encryption and decryption, asymmetric algorithms make use of two different types of keys i) a public-key and ii) a private key (Figure 4.6). The public key is used for encryption and it is shared between other users so messages are encrypted with the public key of the receiver. The private key is used for decryption and it is known only to the owner keeping the secret of the encrypted message to the receiver.

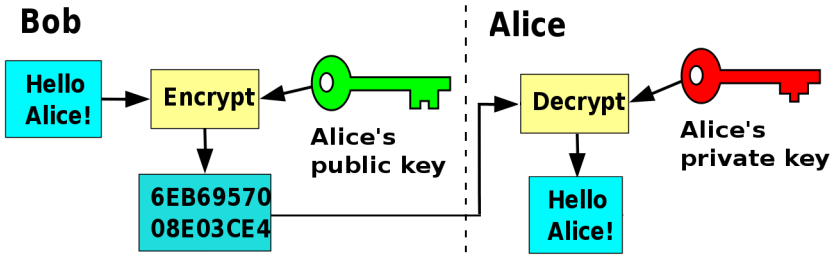


Figure 4.6: Encryption/Decryption scheme in asymmetric cryptography

This kind of cryptographic algorithms can also be used for digital signatures (Figure 4.7) in which keys are applied the other way around, a message is signed with the private key of the sender and is verified with the public key of the sender.

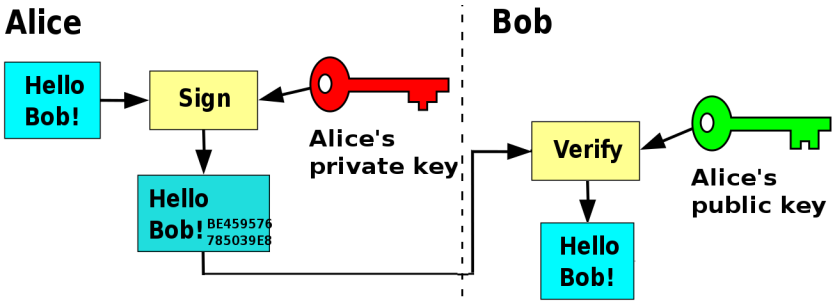


Figure 4.7: Digital signature scheme in asymmetric cryptography

RSA encryption(4.3) and decryption(4.4) need to perform just two operations, but the operational complexity of these operations is as costly as the factoring problem. So the security of the algorithm remains unbroken until factoring problem could be rapidly solved for large numbers.

The operations applied in RSA involve three large positive integers: i) the encryption key e, ii) the decryption key d, and a number use as a part of the public or private key depending on the operation n . There is not restriction for e and d but n must be value such that $0 \leq m \leq n$ where m is the value of the message to encrypt. Such that for all the previously described integers the following equation holds:

$$(m^e)^d \equiv m \bmod n \tag{4.1}$$

$$(m^d)^e \equiv m \bmod n \tag{4.2}$$

The previous values e, d and n involved in the computations cannot be any value but they have to be calculated based on the following rules.

1. n is obtained as the product of two very large randomly chosen prime numbers p and q
2. e can be chosen as any value following $1 \leq e \leq \lambda(n)$ where λ represents the Carmichael's function[14].
3. d is computed based on n and d following this equation $d = \frac{1 \ mod \ \lambda(n)}{e}$

Encryption operation is calculated given a plaintext m as:

$$c \ = \ m^e \ \texttt{mod} \ n \tag{4.3}$$

Decryption operation does the same as (4.3) for the encryption but applying the power of the encrypted message to the decryption key:

$$m \ = \ c^d \ \texttt{mod} \ n \tag{4.4}$$

4.2.2 RTL implementation of the original algorithm

As the previous subsection 4.2.1 defines, RSA is based on a modular exponentiation operation. In order to compute efficiently, this operation square-and-multiply algorithm is used in the original RTL design of the RSA algorithm that we will protect with residue checker.

Square-and-multiply also called exponentiation by squaring or binary exponentiation, the method examines the bits of the exponent to determine which powers are computed and then it divides the operations in powers of two and multiplications (4.5).

$$x^n = \begin{cases} x\,(x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even.} \end{cases} \tag{4.5}$$

The RTL of the RSA algorithm to modified was taken from the Trojan benchmarks' code BasicRSA-Tx00 in TrustHub[41]. The design of this code is done in VHDL and follows the following scheme represented in figure 4.8. In the depicted schema from figure 4.8, each block represents a VHDL component and its instances are represented within a block. The input/output signals of the *modsqr* block are not drawn for simplicity, but it has the same inputs/outputs as the *modmultiply* block as they both instantiate the same component.

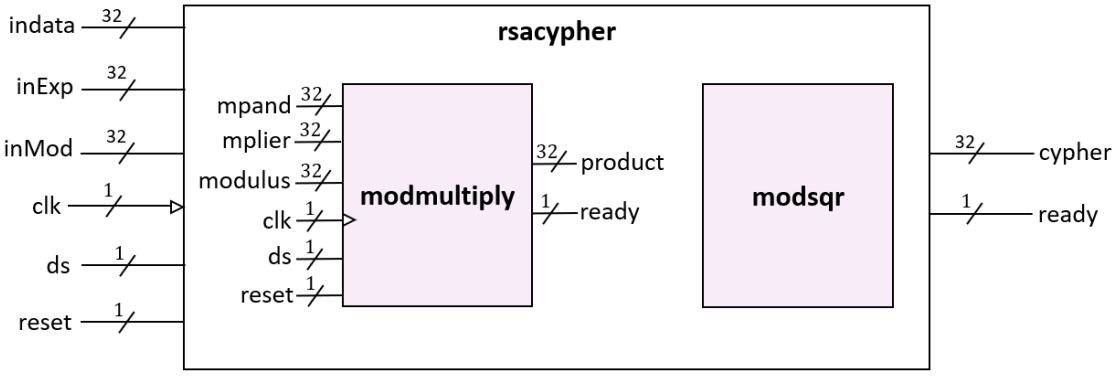


Figure 4.8: RSA schema of the components from the original RTL

The *rsacypher* is the main module receiving the plaintext or encrypted text in *indata*, the encryption or decryption key in *inExp*, the module in *inMod* along with some control signals. This module controls the modular multipliers across the value it pass them, and it collects the results they return.

The *modmultiply* and *modsqr* blocks instantiate the modular multiplier component and they use it to produce products and to take care of squaring operations, respectively. These blocks are the ones actually computing RSA. For that reason, most of the register protected in this design are included inside this component.

```
1 product = prodreg4
2 if (mpreg[0] == '1')
3   prodreg1 = prodreg + mcreg
4 else
5   prodreg1 = prodreg
6 // Subtract modulus and subtract modulus * 2
7 prodreg2 = prodreg1 - modreg1
8 prodreg3 = prodreg1 - modreg2
9 // Negative results mean that we subtracted too much
10 modstate = prodreg3(mpwid+1) & prodreg2(mpwid+1);
11 // Select the correct modular result and copy it
12 if (modstate == '11')
13   prodreg4 = prodreg1
14 else if (modstate == '10')
15   prodreg4 = prodreg2
16 else
17   prodreg4 = prodreg3
18 // Subtract the modulus from the shifted multiplicand
19 mcreg1 <= mcreg - modreg1;
20 // Select the correct modular value and copy it.
21 if (mcreg1(32) == '1')
22   mcreg2 = mcreg
23 else
24   mcreg2 = mcreg1
```

Listing 4.1: Main operations of RSA within the modular multiplier component

Listing 4.1 shows the internal signals use in the modular multiplier components and the main operations applied to calculate RSA. These signals are controlled by the bits of a multiplier and a multiplicand values based on the *indata* signal of the *rsacypher* that represents the message to encrypt/decrypt.

The modular multiplier operates under the Shift-and-Add algorithm to efficiently multiply integers as the square-and-multiply algorithm requires. The algorithm first shifts the multiplicand (i.e. *mpand*) value for each bit of the multiplier (i.e. *mplier*) storing these values in *mcreg*. Then for each bit of the multiplier which value is one, the shifted multiplicand is added to the product as operation in line 3 shows in listing 4.1. Subtractions are computed to make sure the product is always expressed as a remainder, lines 7 and 8 in listing 4.1. The most significant bits of these subtractions, line 10 in listing 4.1, are used to the value of the product, lines 12 to 17. The algorithm ends after going across all bits set to one in the multiplier.

4.2.3 Modifications to the design

RSA operations allow applying the original definition of residue checker to any module as explained in the previous subsection 4.1.1.

The protection mechanism of the operations is based on duplicating the signals of the original algorithm and applied the same operations based on the modules (Listing 4.2). These have been done by partially duplicating the functionality of the algorithm but with a much smaller amount of signals and bits per signal. A block representation of these changes on the design is shown in figure 4.9.

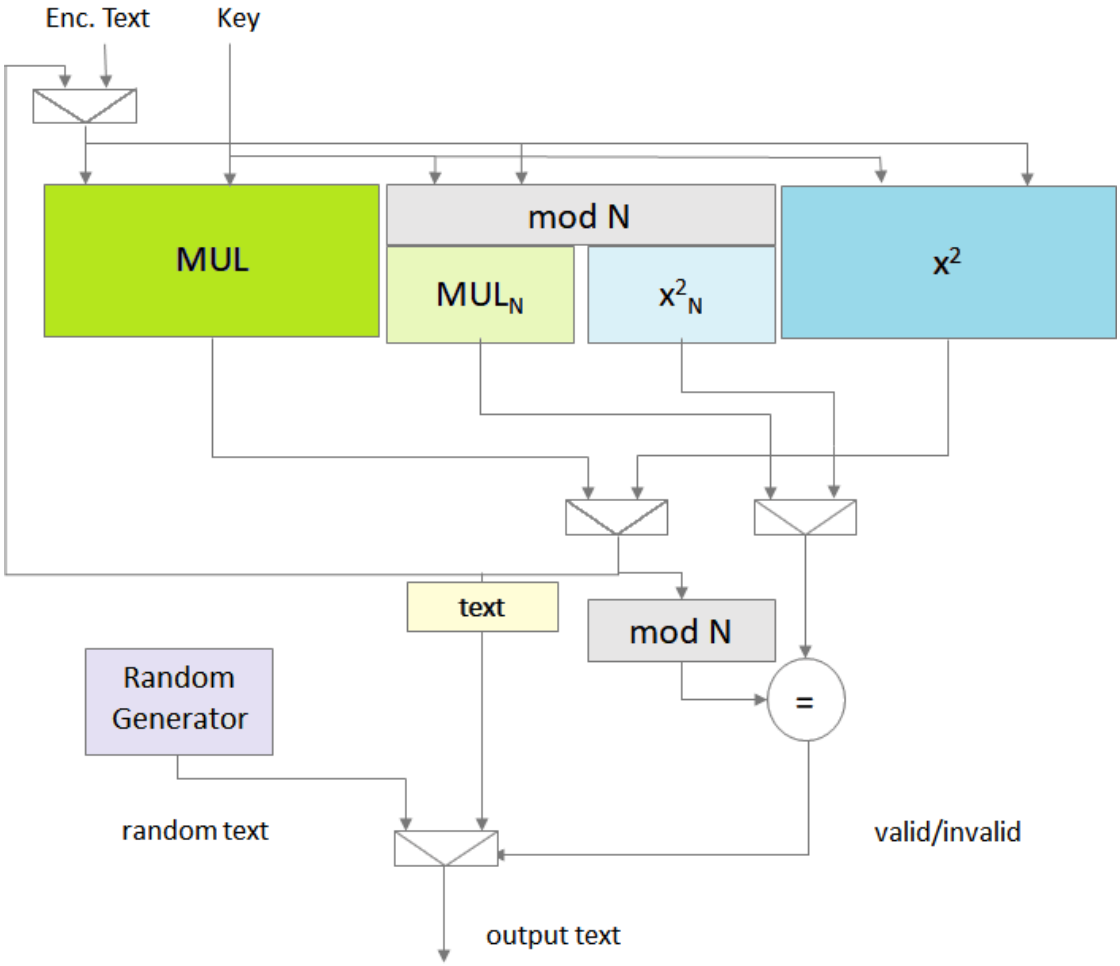


Figure 4.9: RSA schema of the modifications added to the original design

In spite of the operations following the modular-arithmetic properties, there are some issues to take into account when they have to be protected like the possible negative results of a subtraction solved like in lines 7 to 10. Additionally, there are some signals that can not be calculated from the replicated modules, lines 35 to 43. The module has been calculated based on the operations that VHDL provides for the modulo (*mod*) and the remainder (*rem*).

```
1 // ------------ Protection of signals with residue checker ------------
2 if (mpreg(0) == '1')
3   prodreg1_rc = (prodreg_rc + mcreg_rc) mod MOD_VALUE
4 else
5   prodreg1_rc = prodreg_rc
6
7 if (prodreg1_rc >= modreg1_rc)
8   prodreg2_rc = (prodreg1_rc - modreg1_rc) mod MOD_VALUE
9 else
10  prodreg2_rc = (MOD_VALUE + (prodreg1_rc - modreg1_rc))
11
12 if (prodreg1_rc >= modreg2_rc)
13  prodreg3_rc = (prodreg1_rc - modreg2_rc) mod MOD_VALUE
14 else
15  prodreg3_rc = (MOD_VALUE + (prodreg1_rc - modreg2_rc))
16
17 if (modstate == '11')
18  prodreg4_rc = prodreg1_rc
19 else if (modstate == '10')
20  prodreg4_rc = prodreg2_rc
21 else
22  prodreg4_rc = prodreg3_rc
23
24 if (mcreg_rc >= modreg1_rc)
25  mcreg1_rc = (mcreg_rc - modreg1_rc) mod MOD_VALUE
26 else
27  mcreg1_rc = (MOD_VALUE + (mcreg_rc - modreg1_rc));
28
29 if (mcreg1(32) == '1')
30  mcreg2_rc = mcreg_rc
31 else
32  mcreg2_rc = mcreg1_rc
33
34 // Copy values from the original algorithm for mcreg and mpreg as use division
35 if (signed(mcreg) >= 0)
36  mcreg_rc = (signed(mcreg) mod MOD_VALUE)
37 else
38  mcreg_rc = (((signed(mcreg) rem MOD_VALUE) + MOD_VALUE) mod MOD_VALUE)
39
40 if (signed(mpreg) >= 0)
41  mpreg_rc = (signed(mpreg) mod MOD_VALUE)
42 else
43  mpreg_rc = (((signed(mpreg) rem MOD_VALUE) + MOD_VALUE) mod MOD_VALUE);
```

Listing 4.2: Pseudo-code of the protected operations of RSA within the modular multiplier component

The verification of the signals values to corroborate that there is no change on the circuit operation is shown in listing 4.3. All signals can be verified to check if there is an error. However, as the algorithms iterates over all values and all the modifications pass through the last signal (i.e. *prodreg4*) the checks can be limited to that signal reducing the overhead in the hardware of this protection mechanism.

```
1 // ------------ Verification of signals with residue checker ------------
2 if(prodreg4_mod != prodreg4_rc) -> ERROR!
3 if(prodreg3_mod != prodreg3_rc) -> ERROR!
4 if(prodreg2_mod != prodreg2_rc) -> ERROR!
5 if(prodreg1_mod != prodreg1_rc) -> ERROR!
6 if(prodreg_mod != prodreg_rc) -> ERROR!
7 if(modreg2_mod != modreg2_rc) -> ERROR!
8 if(modreg1_mod != modreg1_rc) -> ERROR!
9 if(mcreg2_mod != mcreg2_rc) -> ERROR!
10 if(mcreg2_mod != mcreg2_rc) -> ERROR!
11 if(mcreg1_mod != mcreg1_rc) -> ERROR!
12 if(mcreg_mod != mcreg_rc) -> ERROR!
13 if(mpreg_mod != mpreg_rc) -> ERROR!
```

Listing 4.3: Pseudo-code of the verification of signals using residue checker in RSA within the modular multiplier component

The *rsacypher* module has been also protected even though most of its signals are one-bit signals used to control the flow of the algorithm. No code is shown as it follows the same mechanism shown in listing 4.2 and 4.3 for verification.

4.3 Protection of SHA applying residue checking

4.3.1 Algorithm

SHA is a cryptographic hash function which initials stands for Secure Hash Algorithm. SHA has multiple variants of the same algorithms changing the size of the keys to encrypt, the construction of the resulting hash and the number of construction rounds it uses making each specific algorithm more or less secure. It has been published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS)[42].

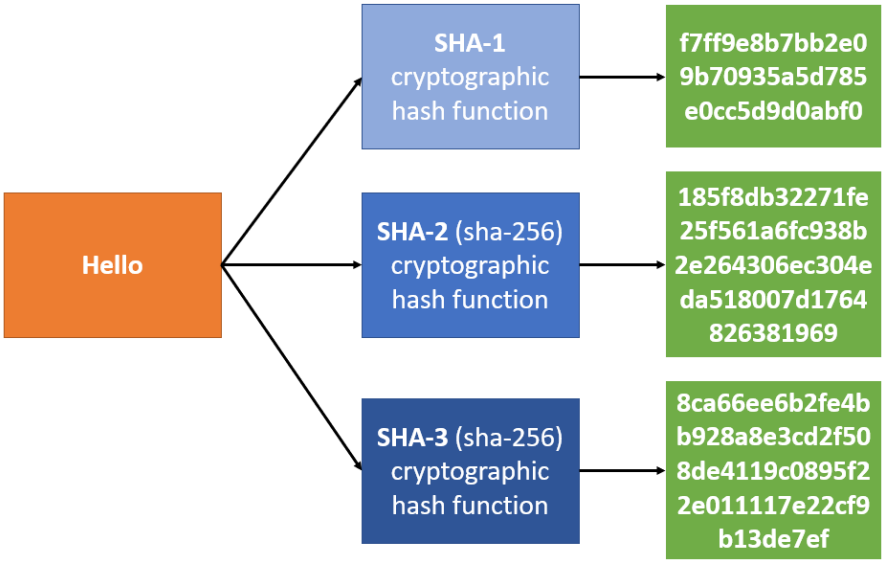


Figure 4.10: Application cryptographic hash functions from the SHA family

SHA function are use for many applications like i) file and messages verification, ii) digital signatures like SSL certificates, iii) password verification or data identifier .

4.3.2 RTL implementation of the original algorithm

From all the SHA functions, the RTL design to apply our protection mechanism with residue checker uses SHA-1. SHA-1 generates a hash function with a length of 160-bit and it can be applied to messages with no length restriction. This algorithm has been discovered to be not secure anymore[5] so the standard was no longer approved for most cryptographic applications after that. However, it can be used to probe that residue checker can protect SHA functions against attack as it is explained in subsection 3.1.

SHA-1 algorithm divides the original plaintext into chunks of 512-bits, 64 characters, and iterates eighty times performing bitwise operations and permutations on this chunk. Then, if the initial message is bigger it takes the next chunks and does the same operations but applying the hash value obtained from the previous chunk to the next chunk and so on. The algorithm finish after going through all the chunks producing a single hash value that was a combination of all the iterations over the message.

The RTL has been developed as a part of the project in the Processor Design subject from MIRI at UPC. The design of the *shacrypto* component has been based on a pseudo code of the SHA-1 algorithm[55]. The design used (Figure 4.11) is defined based on three main components: i) the cryptographic module called *shacrypto*, ii) a MIPS16 processor (i.e. *MIPS16*) and iii) a main data memory (i.e *SRAM*). In spite of the three modules, the modifications to protect the design are only done within the *shacrypto* that is the one defining the SHA-1 function.

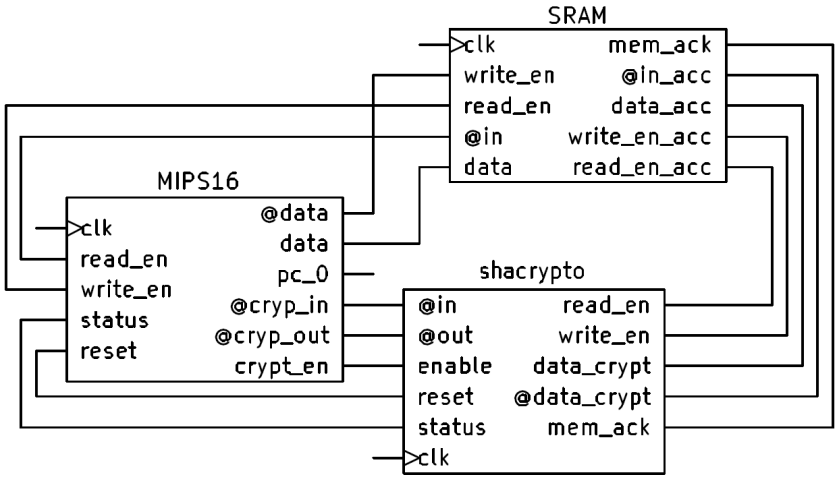


Figure 4.11: SHA-1 schema of the components from the original RTL

The *shacrypto* component has two input signals from *MIPS16*. There is a *enable* input signal to the module coming from the processor to indicate that the module has to start computing an SHA-1 operation. Another input entry, from the processor to the module, is the address in memory where the data to encrypt is located, @*in*.

The outputs are the address in memory where the encoded data has been placed, called @*out* in the picture of figure 4.11. Also, there are two signals to indicate if the module wants to read or write from memory called *read_en* and *write_en*, respectively. Moreover, there is a one-bit output signal that indicates the end of the encoding process named *done*.

Additionally, the cryptography accelerator is connected with main memory (SROM) by the entries called *data* and @*data* that, respectively, indicate the data to encrypt and the address where this data is located in memory.

The *shacrypto* module instantiates a *pre-processor* that is in charge of applying the initial modifications to raw data received from memory, so as the SHA-1 algorithm can be applied. These modifications are:

These modifications are:

- **Append** a bit with value one to the raw message.
- **Append** k zeroes starting from that one bit so that the equation $l + 1 + k \equiv 448 \, mod \, 512$ is satisfied.
- **Append** a 64-bit block (l) containing the length of the original message.

In figure 4.12 the definition of the pre-processing module can be seen. This module is fed with the clock, a reset signal, the length of the raw data, a 512-bit chunk containing the raw data, and finally an enable signal. After one cycle, the module produces the processed data, which is output through *data_out* and a ready signal.

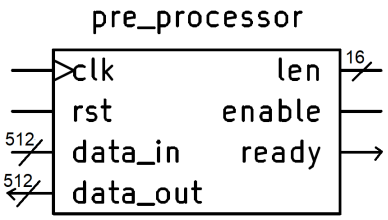


Figure 4.12: *Pre-processor* component instantiate within the *shacrypto* component

Listing 4.4 describes using a pseudo-code the operations done on the RTL of the *shacrypto* component to perform SHA-1 function.

Figure 4.13 shows the components inside the *shacryp* component. This cryptographic component has a cache memory to store the data to be encoded and the hash key. It also has multiple registers, in this case, ninety registers have been defined. The registers contain the data to perform the cryptography encoding and the hash key.

Then, there is a counter to perform the eighty rounds needed by the SHA-1 algorithm. There are four sets of rounds that operate over the values applying different logic functions. For that reason, there are two multiplexers to compute all possible values in parallel and then obtain the correct computed value based on the current round. Finally, the values obtained by the multiplexer and other values from the registers are added to obtain a new value needed on next rounds. In the end, the values of the registers storing a,b,c,d and e are updated as the algorithm indicates. This final storing is not represented in figure 4.13 to reduce the number of connections on the draw.

```
for chunk 0 to NUM_CHUNKS
  for counter_val in 0 to 79
    if (counter_val >= 0 AND counter_val <= 19)
      f = (b and c) or ((not b) and d)
      k = 0x5A827999
    else if (counter_val >= 20 AND counter_val <= 39)
      f = b xor c xor d
      k = 0x6ED9EBA1
    else if (counter_val >= 40 AND counter_val <= 59)
      f = (b and c) or (b and d) or (c and d)
      k = 0x8F1BBCDC
    else if (counter_val >= 60 AND counter_val <= 79)
      f = b xor c xor d
      k = 0xCA62C1D6

    temp = (a leftrotate 5) + f + e + k + w[i]
    e = d
    d = c
    c = b leftrotate 30
    b = a
    a = temp

  if (counter_val == 79) // Save this chunk to the result
    h0 = h0 + a
    h1 = h1 + b
    h2 = h2 + c
    h3 = h3 + d
    h4 = h4 + e

hh = {a, b, c, d, e} //Produce the final hash as a 160-bit number
```

Listing 4.4: Pseudo-code of the operations performed by the SHA-1 function

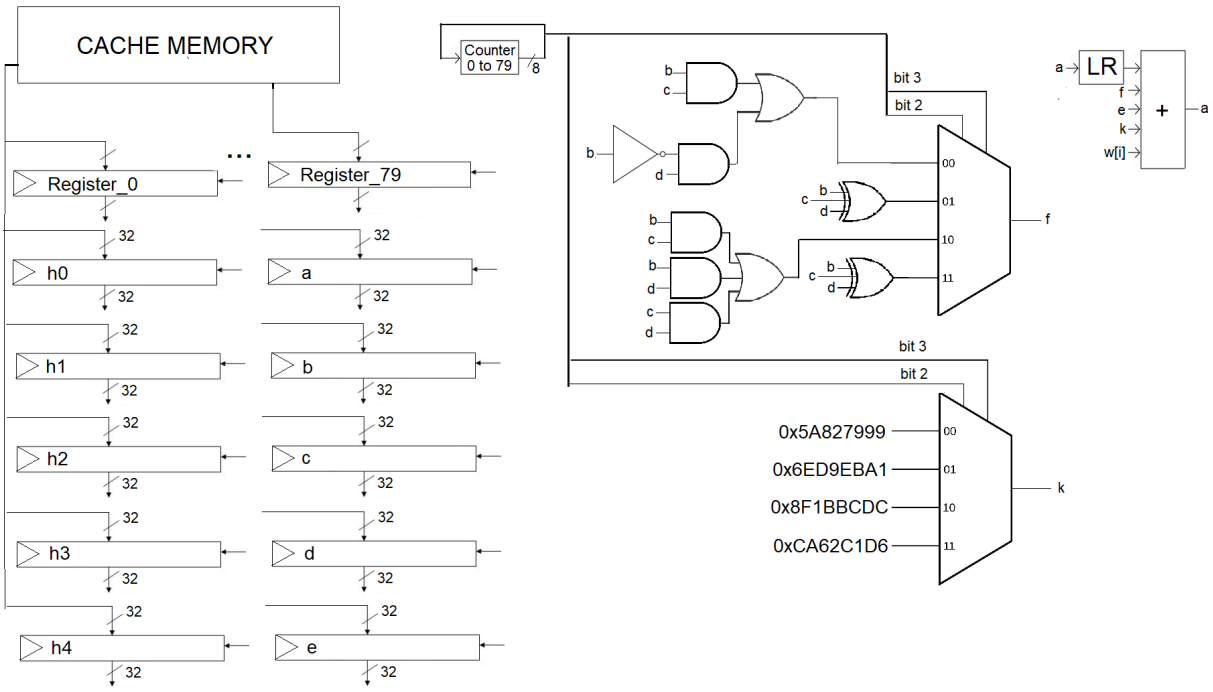


Figure 4.13: Internal overview of the SHA-1 function based on the RTL design

4.3.3 Modifications to the design

The SHA-1 algorithm is mostly composed by bitwise operations (e.g. AND, OR, XOR, NOT) so residue checker cannot be directly applied as this operations do not follow the modular arithmetic properties to can use residue checker for any module. For that reason, as it is explained in subsection 4.1.1 residue checker has been applied using power-of-two modules that it is basically to protect the last n bytes when the module 2^n is used.

Listing 4.5 shows a pseudo-code of the protected operations for SHA-1 where the replicated signals are denoted as the name of the original signals concatenate with the suffix *_rc*. But, these replicated signals has a size of n bits where $0 < n < 32$ as the original size of the protected signal is 32 bits.

```
for counter_val in 0 to 79
  if (counter_val >= 0 AND counter_val <= 19)
    f_rc = (b_rc and c_rc) or ((not b_rc) and d_rc)
    k_rc = 0x5A827999 // Take last n bits of this constant
  else if (counter_val >= 20 AND counter_val <= 39)
    f_rc = b_rc xor c_rc xor d_rc
    k_rc = 0x6ED9EBA1 // Take last n bits of this constant
  else if (counter_val >= 40 AND counter_val <= 59)
    f_rc = (b_rc and c_rc) or (b_rc and d_rc) or (c_rc and d_rc)
    k_rc = 0x8F1BBCDC // Take last n bits of this constant
  else if (counter_val >= 60 AND counter_val <= 79)
    f_rc = b_rc xor c_rc xor d_rc
    k_rc = 0xCA62C1D6 // Take last n bits of this constant

  temp_rc = (a leftrotate 5) + f_rc + e_rc + k_rc + w[i]
  e_rc = d_rc
  d_rc = c_rc
  c_rc = b leftrotate 30
  b_rc = a_rc
  a_rc = temp
```

Listing 4.5: Pseudo-code of the protected operations executed in SHA-1 function

The code verifying that these signals are correctly protected is shown in listing 4.6. This code is quite similar to the one for the protecting the other cryptographic algorithms (listing 4.3 and 4.12) as it only checks if the protected signals are equal to the protected part from the original signals and if throws an error in case the condition is not satisfied.

```
if(a_rc != a[(MOD_SIZE - 1)..0]) -> ERROR!
if(b_rc != b[(MOD_SIZE - 1)..0]) -> ERROR!
if(c_rc != c[(MOD_SIZE - 1)..0]) -> ERROR!
if(d_rc != d[(MOD_SIZE - 1)..0]) -> ERROR!
if(e_rc != e[(MOD_SIZE - 1)..0]) -> ERROR!
if(f_rc != f[(MOD_SIZE - 1)..0]) -> ERROR!
if(k_rc != k[(MOD_SIZE - 1)..0]) -> ERROR!
if(temp_rc != temp[(MOD_SIZE - 1)..0]) -> ERROR!
```

Listing 4.6: Pseudo-code of the verification operations executed in SHA-1 function

Additionally, it has also been applied to the design of SHA-1 the rotary residue checker defined in subsection 4.1.1 with the objective of improving the detection rate on this algorithm as the limitation of only using power-of-two in residue checker always keep some unprotected bits and this fact detriments our protection system for cases in which the number of protected bits is very low.

The pseudo-code of the rotary residue checker application to protect the operations of SHA-1 function is shown in listing 4.7. The rotary residue checker replicates the operation mostly in the same way it was done without the rotation mechanism that we show in listing 4.5. However, after n rounds the protected bits are changed and due to that, the protected bits must be copied from the original signals before using them.

```
K_HASH_ROUND_1_19 = 0x5A827999
K_HASH_ROUND_20_39 = 0x6ED9EBA1
K_HASH_ROUND_40_59 = 0x8F1BBCDC
K_HASH_ROUND_60_79 = 0xCA62C1D6

for counter_val in 0 to 79
  if (counter_val >= 0 AND counter_val <= 19)
    f_rc = (b_rc and c_rc) or ((not b_rc) and d_rc)
    k_rc = K_HASH_ROUND_1_19[(iniBit + MOD_SIZE - 1..)iniBit]
  else if (counter_val >= 20 AND counter_val <= 39)
    f_rc = b_rc xor c_rc xor d_rc
    k_rc = K_HASH_ROUND_20_39[(iniBit + MOD_SIZE - 1..)iniBit]
  else if (counter_val >= 40 AND counter_val <= 59)
    f_rc = (b_rc and c_rc) or (b_rc and d_rc) or (c_rc and d_rc)
    k_rc = K_HASH_ROUND_40_59[(iniBit + MOD_SIZE - 1..)iniBit]
  else if (counter_val >= 60 AND counter_val <= 79)
    f_rc = b_rc xor c_rc xor d_rc
    k_rc = K_HASH_ROUND_60_79[(iniBit + MOD_SIZE - 1..)iniBit]

  temp_rc = (a leftrotate 5) + f_rc + e_rc + k_rc + w[i]
  e_rc = d_rc
  d_rc = c_rc
  c_rc = (b leftrotate 30)[(iniBit + MOD_SIZE - 1..)iniBit]
  b_rc = a_rc
  a_rc = temp[(iniBit + MOD_SIZE - 1..)iniBit]

  // Copy values when protected bits are rotated after 3 iterations
  if((counter_val mod 3 == 0) AND (counter_val >= 3) AND (REG_WIDTH > MOD_SIZE))
    if (rst == 1)
      new_iniBit = 0
    else
      // Take next n bits there are still n or more most significant bits to protect
      if ((iniBit + MOD_SIZE) < (REG_WIDTH - MOD_SIZE))
        new_iniBit = iniBit + MOD_SIZE
      else // Restart index if the most significant bits were protected on last round
        if ((iniBit == (REG_WIDTH - MOD_SIZE)))
          new_iniBit = 0
        else // Protected the most significant n bits if not protected
          new_iniBit = REG_WIDTH - MOD_SIZE;
    // Copy new bits to protect keeping the bits already protected if possible
    if ((new_iniBit > iniBit) AND ((new_iniBit + MOD_SIZE) == REG_WIDTH)
    AND ((REG_WIDTH mod MOD_SIZE) != 0))
      a_rc = {a[REG_WIDTH..(REG_WIDTH-(new_iniBits-iniBits))], a_rc >> (new_iniBits-iniBits)}
      b_rc = {b[REG_WIDTH..(REG_WIDTH-(new_iniBits-iniBits))], b_rc >> (new_iniBits-iniBits)}
      c_rc = {c[REG_WIDTH..(REG_WIDTH-(new_iniBits-iniBits))], c_rc >> (new_iniBits-iniBits)}
      d_rc = {d[REG_WIDTH..(REG_WIDTH-(new_iniBits-iniBits))], d_rc >> (new_iniBits-iniBits)}
      e_rc = {e[REG_WIDTH..(REG_WIDTH-(new_iniBits-iniBits))], e_rc >> (new_iniBits-iniBits)}
    else if ((new_iniBit < iniBit) AND ((REG_WIDTH div 2) > MOD_SIZE))
      a_rc = {a_rc << (iniBit - new_iniBit), a[(iniBit - new_iniBit)..new_iniBit]
      b_rc = {b_rc << (iniBit - new_iniBit), b[(iniBit - new_iniBit)..new_iniBit]
      c_rc = {c_rc << (iniBit - new_iniBit), c[(iniBit - new_iniBit)..new_iniBit]
      d_rc = {d_rc << (iniBit - new_iniBit), d[(iniBit - new_iniBit)..new_iniBit]
      e_rc = {e_rc << (iniBit - new_iniBit), e[(iniBit - new_iniBit)..new_iniBit]
    else //Copy all bits from the original signals
      a_rc = a[(new_iniBit + MOD_SIZE - 1)..MOD_SIZE]
      b_rc = b[(new_iniBit + MOD_SIZE - 1)..MOD_SIZE]
      c_rc = c[(new_iniBit + MOD_SIZE - 1)..MOD_SIZE]
      d_rc = d[(new_iniBit + MOD_SIZE - 1)..MOD_SIZE]
      e_rc = e[(new_iniBit + MOD_SIZE - 1)..MOD_SIZE]
    iniBit = new_iniBit // Set the initial bit to the new protected bit after copy
```

Listing 4.7: Pseudo-code of the protection in SHA-1 with rotary residue checker

From listing 4.7, we can see that the rotary mechanism does not rotate only on one direction taking MOD_SIZE bits to the left in each rotation, where MOD_SIZE is the number of bits to protect. This has been done because it is not possible to protect non-consecutive bits as it is not possible to make the residue checker work this way. For that reason, the rotary mechanism protects all the possible chunks of MOD_SIZE bits going from the less significant to the most significant bit until the next chunk is smaller than MOD_SIZE. In this particular, case happens the rotary mechanism act in an inverse way taking MOD_SIZE bits from the most significant to the less significant bit and then it resets the rotary mechanism to start once again from bit zero. This has been done because protecting only the missing most significant bits, in case there are less that MOD_SIZE, would lead to less protection.

Figure 4.14 shows an example of how this rotation mechanism would work using a module value non-multiple of the register width for the next three rounds. Fourth round and the following are not represented but they will do the same restarting the initial protected bit in the fourth round behaving as in the first round and so on.

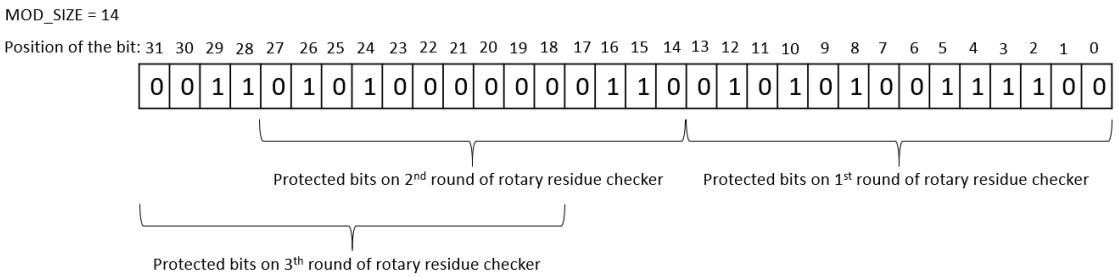


Figure 4.14: Example of the rotary residue checker protecting 14 bits

The code verifying that these signals are correctly protected is the same as the one for the residue checking without the rotary mechanism shown in listing 4.6, but taking into account that now the bits that have to be checked change depending on the rotation as listing 4.8.

```
1 // ------------ Verification of signals with rotary residue checker ------------
2 if(a_rc != a[(new_iniBit + MOD_SIZE - 1..)new_iniBit]) -> ERROR!
3 if(b_rc != b[(new_iniBit + MOD_SIZE - 1..)new_iniBit]) -> ERROR!
4 if(c_rc != c[(new_iniBit + MOD_SIZE - 1..)new_iniBit]) -> ERROR!
5 if(d_rc != d[(new_iniBit + MOD_SIZE - 1..)new_iniBit]) -> ERROR!
6 if(e_rc != e[(new_iniBit + MOD_SIZE - 1..)new_iniBit]) -> ERROR!
7 if(f_rc != f[(iniBit + MOD_SIZE - 1..)iniBit]) -> ERROR!
8 if(k_rc != k[(iniBit + MOD_SIZE - 1..)iniBit]) -> ERROR!
9 if(temp_rc != temp[(iniBit + MOD_SIZE - 1..)iniBit]) -> ERROR!
```

Listing 4.8: Pseudo-code of the verification operations performed by the SHA-1 function using rotary residue checker

With the extra control part in the case of the rotary mechanism, we can improve the detection of faults using residue checker on this concrete algorithm as the evaluation, chapter 5, explains in more detail.

4.4 Protection of AES applying residue checking

4.4.1 Algorithm

AES stands for Advanced Encryption Standard but it is also known by the name of Rijndael. This cryptographic algorithm was defined in 1998 by and it was established as a Federal Information Processing Standard (FIPS) in 2001[52] by the U.S. National Institute of Standards and Technology (NIST).

AES is a symmetric key algorithm like RSA so it can be used for encryption and decryption using the same key. More information about symmetric key algorithms can be checked on the previous section 4.2.1 for RSA.

AES algorithm is based on a set of operation performed over the message to be encrypted or decrypted applying the cipher key. The steps performed by AES are the next ones:

1. KeyExpansion operation. This operation derives round keys from the cipher key using a schedule defined by the authors.

2. AddRoundKey operation is performed on the initial round of the algorithm. AddRoundKey applies the exclusive or operation for each byte of the state[1] with a block of the round key (Figure 4.15).

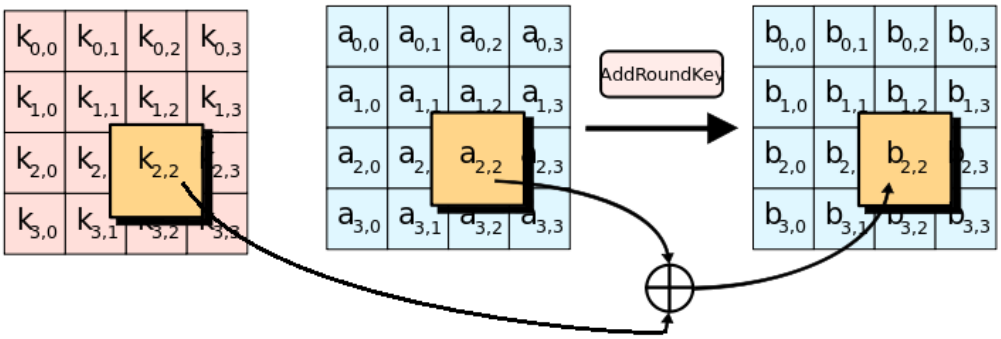


Figure 4.15: Add round key operation[54]

3. Round key operations is a set of operations applied sequentially during 9, 11 or 13 rounds depending on if the key size is of 128, 192 or 256 bits, respectively. This set of operations includes the following one concrete operations:

   - SubBytes operation. This operation replaces each byte by another according to a predefined lookup table (Figure 4.17).
   - ShiftRows operation. This operation shifts the last three rows of the state to a certain position (Figure 4.18).
   - MixColumns operation combines the bytes of each column of bytes by applying the exclusive or operation to each byte with a fixed polynomial (Figure 4.16).

[1]state: Message to encrypt/decrypt using AES. It can be the original message or a modification of it depending on the round.

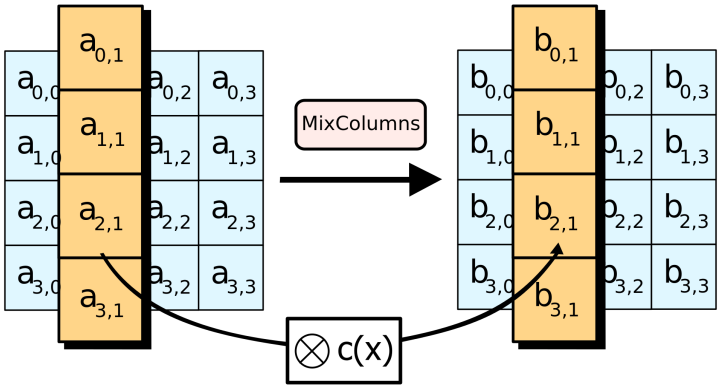


Figure 4.16: Mix columns operation[54]

- AddRoundKey operation (Figure 4.15)

4. The following operation are performed on the final round being the 10, 12 or 14 round depending on the key length being 128, 192 or 256 bits, respectively.

   - SubBytes operation (Figure 4.17)

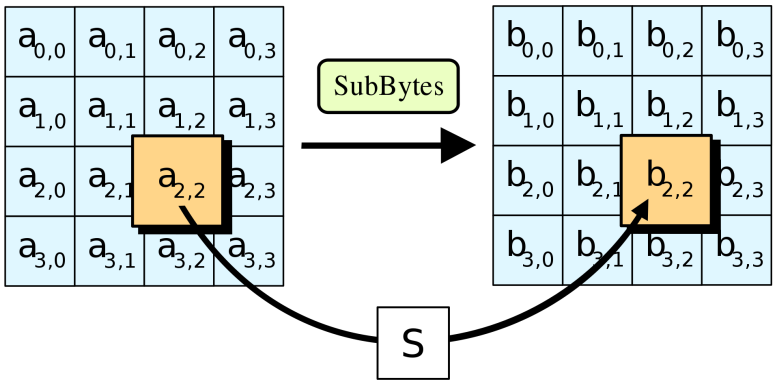


Figure 4.17: Substitution of bytes operation[54]

   - ShiftRows operation (Figure 4.18)

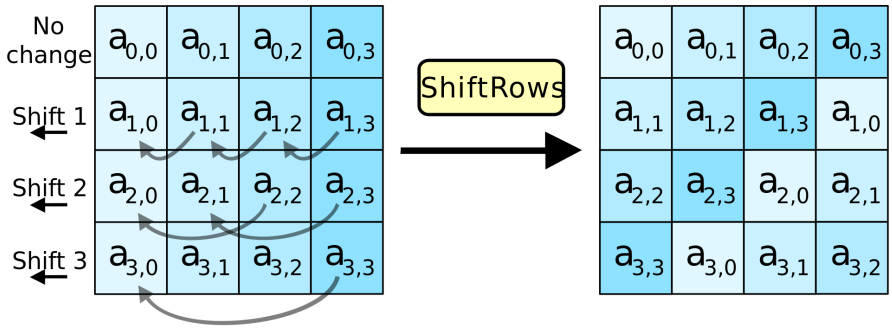


Figure 4.18: Shift rows operation[54]

   - AddRoundKey operation (Figure 4.15)

AES can be defined to be secured for most of the attacks perform to then as the operational complexity to discover its cipher key is huge if it is tried to break using the most simple of the attacks that it is to bruce-force-attack that it tries all the possible key. However other attacks reduce substantially this operational complexity. One example is differential fault analysis that recovers the key in seconds[28]. In spite of differential fault attacks, AES is included in the instructions set of modern processors such as Intel or AMD.

4.4.2 RTL implementation of the original algorithm

AES algorithm can be implemented for different key sizes, in this case, the chosen design uses the smallest key possible that it has 128 bits.

The RTL design chosen is the Trojan free design provided in the AES-T100 Trojan's benchmark from TrustHub[41]. This design is composed by four components: i) *aes_128*, ii) *expand_key_128*, iii) *one_round* and iv) *final_round*. These components from the RTL design to be modified are depicted in 4.19.

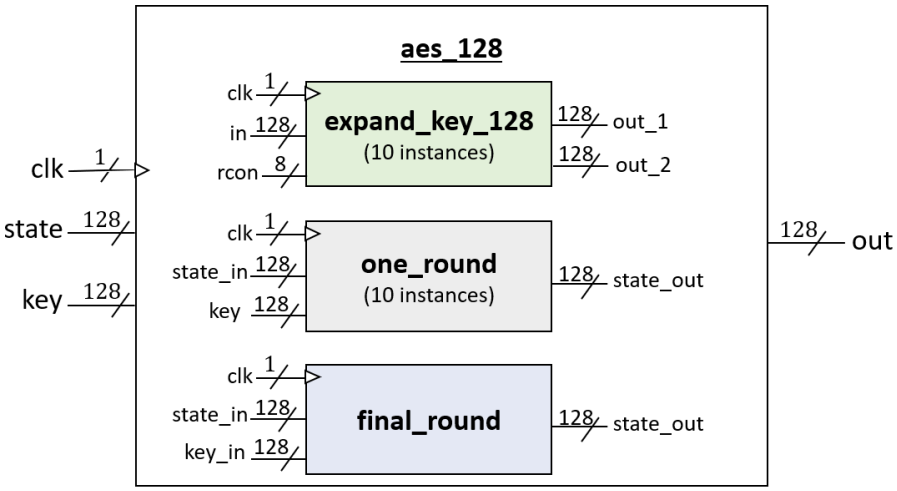


Figure 4.19: AES schema of the components from the original RTL

The *aes_128* is the external component controlling the input message to which apply the AES algorithm called *state* and the cipher key called *key*. It instantiates the other components and passes these inputs to them in order to compute the AES algorithm defined on the previous subsection 4.4.1. Also, it runs the $addRoundKey$ operation using base on the input *key* and *state*.

The *expand_key_128* component is instantiated ten times in order to do the key expansion operation that derives a set of keys from the cipher key that is input to *aes*_128. This key expansion is done sequentially as instances need the output of the previous instance in *out*_1 as the *in* input to can perform the key expansion operation. Then, it uses the result obtained from each instance to execute the *addRoundkey* operation defined in point 2 of the algorithm described in subsection 4.4.1.

The *one_round* component is instantiated nine times and it uses the expand key output returned by the *expand_key_128* component, called *out_2* in figure 4.19, as the input of the *state_in* signal to compute all three operations explained in point 3 of the algorithm described in subsection 4.4.1. These operations are executed sequentially as the output of the first instance, called *state_out* is the *state_in* input of the second instance and so on.

The *final_round* is only instantiated once and it perform the operations defined in point 4 of the algorithm described in subsection 4.4.1 having as the *key_in* input the *out_2* from the last instance of *expand_key_128* component and as the *state_in* input the *state_out* output from the last instance of *one_round* component.

4.4.3 Modifications to the design

The design of AES described on the previous subsection 4.4.2 has been modified to apply residue checker in order to protect all the possible operations. However, not all operation can be protected without adding a big overhead to the original design. For that reason, these operations are not duplicated on our protection mechanism. One example of a not protected operation in our implementation for AES is the subBytes operation. This operation is not protected as it needs of a lookup table to do the substitution of bytes and we do not want to replicate all this table. For that reason, only bitwise operations are protected.

As it is previously mentioned bitwise operations are the objective as AES does not have any arithmetic operation that can be protected. This implies that the implementation of the residue checker for this algorithm has the same limitation as to the SHA-1 algorithm so only power-of-two values for the module can be used. And due to this restriction, the protection of the operations is not as high as if non-power of two values had been used unless the modules are big numbers close to 2^{signal_width}.

The initial *addRoundKey* operation cannot be protected as it is directly applies from the input key and state of the main module (i.e. *aes_128*), point 2 of the described algorithm in subsection 4.4.1. Listing 4.9 shows how this operation is performed.

```
1 // AddRoundKey operation applies the exclusive or operation between
2 // the input state and key
3 s0 = state XOR key;
```

Listing 4.9: Pseudo-code of the initial *addRoundKey* operation performed as part of the AES algorithm described in point 2 of the algorithm description subsection 4.4.1

The *subBytes* operation, described in point 3 of the described algorithm in subsection 4.4.1, substitutes the bytes using a lookup table that it is not protected to avoid the overhead in replication. The application of this operation can be seen in the pseudo-code of listing 4.10.

```
1 // Application of the subBytes operation to the input of one_round
2 p = SubBytes(state_in);
```

Listing 4.10: Pseudo-code of the *subBytes* operation performed as part of the rounds of AES algorithm described in point 3 of the algorithm description subsection 4.4.1

The *shiftRows* operation applied in the nine-round operations described in point 3 of the described algorithm in subsection 4.4.1 is performed along with the *mixColumns* operations and the *addRoundKey* operation. Therefore, as these three operations are executed on the same operation, they are protected together. The protection of these operations has been protected by replicating some bits of the original signals and replicating the exclusive or operation of the last three rows with a predefined value as the pseudo-code in listing 4.11 shows.

```
// Get state columns of the state by coping the input to avoid
// possible internal manipulation within the component of the state_in signal
s0_rc = state_in[(96 + MOD_SIZE - 1)..96]
s1_rc = state_in[(64 + MOD_SIZE - 1)..64]
s2_rc = state_in[(32 + MOD_SIZE - 1)..32]
s3_rc = state_in[(MOD_SIZE - 1)..0]

// Divide output of the subBytes operations by cells
p = {p00, p01, p02, p03}
    {p10, p11, p12, p13}
    {p20, p21, p22, p23}
    {p30, p31, p32, p33}

// Perform shiftRows and mixColumns operation to compute the value of the
// state_out signal
z0_rc = p00 XOR p11 XOR p22 XOR p33 XOR k0_rx
z1_rc = p03 XOR p10 XOR p21 XOR p32 XOR k1_rc
z2_rc = p02 XOR p13 XOR p20 XOR p31 XOR k2_rc
z3_rc = p01 XOR p12 XOR p23 XOR p30 XOR k3_rc
```

Listing 4.11: Pseudo-code of the protected *mixColumns* operation performed as part of the rounds of AES algorithm described in point 3 of the algorithm description subsection 4.4.1

The verification of these operations, shown in listing4.11, has been done by checking that the replicated signals have the same value as the original ones. In the same way, as it has been done for the other algorithms, RSA (subsection 4.2.3) and SHA-1 (subsection 4.3.3). Listing 4.12 shows concretely how this verification has been done.

```
// ------------ Verification of signals with residue checker ------------
if (k0[(MOD_SIZE - 1)..0] != k0_rc) -> ERROR!
if (k1[(MOD_SIZE - 1)..0] != k1_rc) -> ERROR!
if (k2[(MOD_SIZE - 1)..0] != k2_rc) -> ERROR!
if (k3[(MOD_SIZE - 1)..0] != k3_rc) -> ERROR!
if (s0[(MOD_SIZE - 1)..0] != s0_rc) -> ERROR!
if (s1[(MOD_SIZE - 1)..0] != s1_rc) -> ERROR!
if (s2[(MOD_SIZE - 1)..0] != s2_rc) -> ERROR!
if (s3[(MOD_SIZE - 1)..0] != s3_rc) -> ERROR!
if (z0[(MOD_SIZE - 1)..0] != z0_rc) -> ERROR!
if (z1[(MOD_SIZE - 1)..0] != z1_rc) -> ERROR!
if (z2[(MOD_SIZE - 1)..0] != z2_rc) -> ERROR!
if (z3[(MOD_SIZE - 1)..0] != z3_rc) -> ERROR!
```

Listing 4.12: Pseudo-code of the verification operations performed by the AES algorithm described in point 3 of the algorithm description subsection 4.4.1

The final round operations described in point 4 of the described AES algorithm in subsection 4.4.1 are protected in a similar way as its corresponds operation was in the previous rounds described in point 3 of AES algorithm description in subsection 4.4.1.

The *subBytes* operation is not protected as it happens in the previous use of this function to avoid lookup tables replication. For that reason, we had omitted the pseudocode performing this operation as it duplicates the listing 4.4.1.

The *shiftRows* and *addRoundKey* operations on the final round are performed together in the RTL design so its protection has been also done in this way as it can be seem in listing 4.13.

```
// Protection of the key signal by coping the input to avoid
// possible internal manipulation within the component of the key_in signal
k0_rc = key_in[(96 + MOD_SIZE - 1)..96]
k1_rc = key_in[(64 + MOD_SIZE - 1)..64]
k2_rc = key_in[(32 + MOD_SIZE - 1)..32]
k3_rc = key_in[(MOD_SIZE - 1)..0]

// shiftRows and addRoundKey operation
z0_rc = {p00, p11, p22, p33} XOR k0_rc
z1_rc = {p10, p21, p32, p03} XOR k1_rc
z2_rc = {p20, p31, p02, p13} XOR k2_rc
z3_rc = {p30, p01, p12, p23} XOR k3_rc

// The state_in signal is protected even though the subBytes operation
// cannot be protected. The protection is done by coping the input
// to avoid possible internal manipulation within the component
s0_rc = state_in[(96 + MOD_SIZE - 1)..96]
s1_rc = state_in[(64 + MOD_SIZE - 1)..64]
s2_rc = state_in[(32 + MOD_SIZE - 1)..32]
s3_rc = state_in[(MOD_SIZE - 1)..0]
```

Listing 4.13: Pseudo-code of the *shiftRows* and *addRoundKey* operations performed as part of the final round of AES algorithm described in point 4 of the algorithm description subsection 4.4.1

The verification of the final round on AES algorithm has been done as listing in lst:aesVerification for the other rounds by confirming that the original signals used in the *shiftRows* and *addRoundKey* operations have the expected value compared to the replicated ones.

5. Evaluation of the work

This chapter describes the details of how the proposal has been evaluated and how the metrics indicated in the earlier section 3.2 have been applied to evaluate the system. Then, it gives the results obtained from this evaluation and it compares them to the calculated theoretical values of how the system should behave.

5.1 Definition of evaluation metrics

The metrics to evaluate are based on two objectives: i) test the detection rate that our proposal provides and ii) calculate the overhead that this implementation put in the algorithms. These two objectives evaluate the feasibility of this technique given the protection needs and overhead limits a system may have.

On one hand, the metrics to obtain the detection rate have been evaluated by testing the system to detect hardware Trojans and to protect against DFA attacks.

The hardware trojans tested are the BasicRSA-T100, BasicRSA-T200 and BasicRSA-T300 benchmarks from TrustHub[41].

DFA attacks are simulated as bit-flip injections into the three tested algorithms. The idea is that if the protection mechanism with residue checking can detect these bit-flips then the system can protect from DFA attacks by non-returning the output obtained from that computation with faults but a random number. This random output ensures that the system does not provide any meaningful information to the attacker, and thus it yields the DFA attack ineffective.

On the other hand, the overhead of the proposal will be measured as: (1) extra amount of power consumption, (2) additional area utilisation, (3) reduction on the maximum operating frequency and (4) the possible affection of the new signals on the timing critical path of the design.

The difference between the power consumption between the initial design and the one including the protection mechanism has been measured based on relative terms as the values have been taken using a tool that estimates these values from a synthesis design. So, these values are not as accurate as measuring the consumption on a physical system but it can still be good to calculate the power overhead.

The difference in area utilisation has been measured in the number of LookUp Tables (LUTs) as the design has been synthesis on two different FPGAs and these components are the common ones used to implement programmable combinational logic.

The maximum operational frequency has been also obtained from a tool calculation based on a synthesis of the design, so it has been also evaluated as a relative term to measure the impact on the frequency after the modifications in the design.

The addition of a worse critical path due to the new signals on the design has been also evaluated based on the synthesis and the idea is to determine if the proposal includes

a new maximum delay path that it can affect the performance of the initial design.

5.2 Performance results

5.2.1 Study of the detection rate

First, hardware Trojan protection has been evaluated obtaining a positive result as all the hardware Trojans from the tested benchmarks are detected.

Then, the DFA attacks have been evaluated running simulations where each execution includes a bit-flip that the detection system will try to detect. All these simulations report if the fault has been detected. This report has been analysed to obtain the detection rate of these techniques against faults.

In order to provide statistically significant results for the simulation, it is usually understood to provide as many simulations as a 99% confidence level at 5% error margin requires. The concrete minimum number of simulations needed[50], n, based on a certain confidence and error margin can be computed as in (5.1) where $z_{\alpha/2}$ is the value of the standard normal for a percentage of confidence of $(1-\alpha)$ and w is the confidence interval width computed based on (5.2).

$$n = \frac{4{z_{\alpha/2}}^2 \times AVF(1 - AVF)}{w^2} \tag{5.1}$$

$$w = 2 \times error\_margin \times AVF \tag{5.2}$$

AVF indicates the Architectural Vulnerability Factor[19] defined as the probability of an error to be visible by the user when a bit flips in a storage cell.

Base on 5.1, we have calculated concrete numbers of the simulation we need for each cryptographic algorithm tested. These values are represented in table 5.1.

Confidence level	**99%**			**95%**		
Error margin	**10%**	**5%**	**1%**	**10%**	**5%**	**1%**
RSA (AVF = 0.51)	638	2551	63756	370	1477	36910
SHA-1 (AVF = 0.53)	589	2354	58846	341	1363	34068
AES (AVF = 0.09)	6709	26839	670951	3885	15538	388429

Table 5.1: Number of simulations for a certain confidence an error margin[50]

The confidence level indicates the confidence interval percentage of the experiments that would return a true result, based on layman's terms. This means that setting the confidence level as high as possible the probability of having the expected result, what it is defined as a true result, on the simulations is higher.

The error margin determines the percentage that the confidence interval can deviate from the theoretical population as many times as the confidence interval sets. So it indicates the precision of the result based on a confidence level. For that reason, this value must be as small as possible. Based on the previous definitions, the ideal value for the confidence interval would be 100% with a 0% of error margin but this would significantly increase the number of simulations to do to an unfeasible number. So we

aim for a tradeoff between the confidence level and error margin with the number of executions or execution time.

Therefore, as the required number of experiments rises with the increase in confidence level and decrease in the error margin, the number of simulations has been calculated for a set of high confidence level and low error margins to check what combination would better fit in precision and execution time.

In the end, the confidence level chosen is 99% while the margin error is 1% to provides the most accurate result within a feasible computation time.

The simulations have been executed in parallel on the Arvei cluster provided by the architecture department at UPC[17]. This cluster is composed of several nodes with different computational power so the execution time varies depending on the node and the load of the server. To give some numbers, the average execution time was of 25 minutes for RSA, 1 hour for SHA-1 and more than 2 hours for AES. In addition to the simulation time, the analysis time must be included. The analysis was automatically performed with a script interpreting the report from the simulation. The analysis execution takes less time going from 4 minutes to analyse RSA report to 57 minutes for AES.

The first analysis done was about the inherent fault tolerance of each algorithm. Due to logical, electrical and temporal masking, not all faults injected will result in a visible change in the output. Thus, the first analysis performed is to measure how many of the injected faults result in a change in the outputs. Figure 5.1 reports the percentage of faults injected that lead to an erroneous output and which not for each of the tested algorithms. From figure 5.1, the difference between the algorithms is significant showing that the percentage of faults tolerate depends directly on the operations and implying that DFA attacks will not be as effective in case the algorithm is able to highly tolerate faults not outputting erroneous errors even if faults have been inserted. Due to the simulation framework, the numbers reported corresponding to logical and temporal masking.

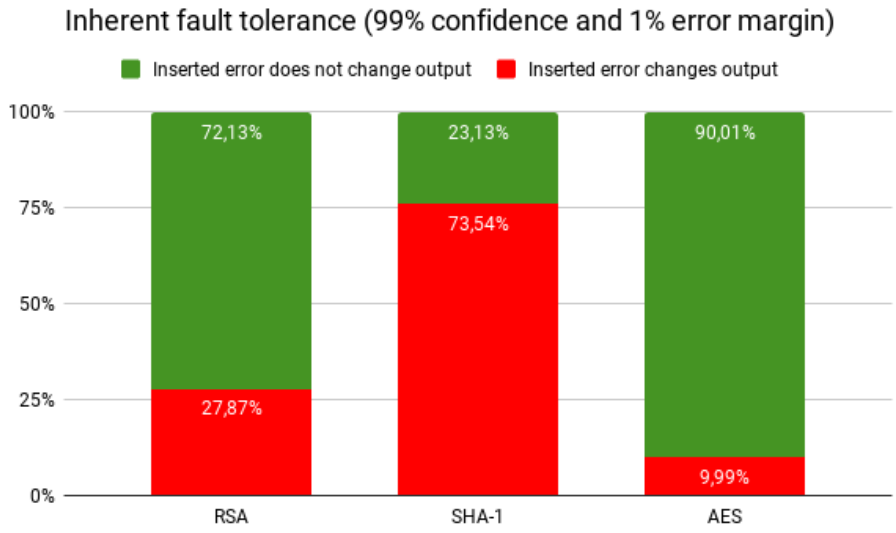


Figure 5.1: Comparison of the inherent fault-tolerance represented by algorithms

As the objective of these simulations is to evaluate how good the proposal protects the algorithm against DFA attacks only the simulation that propagates an error has been taken into account to evaluate the proposed protection system. For that reason the following figures 5.2, 5.3 and 5.4 display the percentage of detected errors according to the value of the modulo use by the residue checker.

Figure 5.2 represents the evaluation of the residue checker for power-of-two modules with the blue squares and the non-power of two values of the module as orange diamonds. These results show that any non-power-of-two value detects all the injected faults while power-of-two value only detects all faults if the signals are completely replicated.

These results were expected as it was previously mentioned when explaining residue checking mechanism in subsection 4.1.1, as power-of-two values, only protect the less n significant bits. Therefore, faults injected on the most significant bits, non-protected, are not detected. Consequently, the detection capabilities of non-power-of-two modules are much higher as without needing to replicate all the bits of the signals it can detect 100% of the faults injected avoiding intentioned erroneous outputs and providing full protection against DFA attacks.

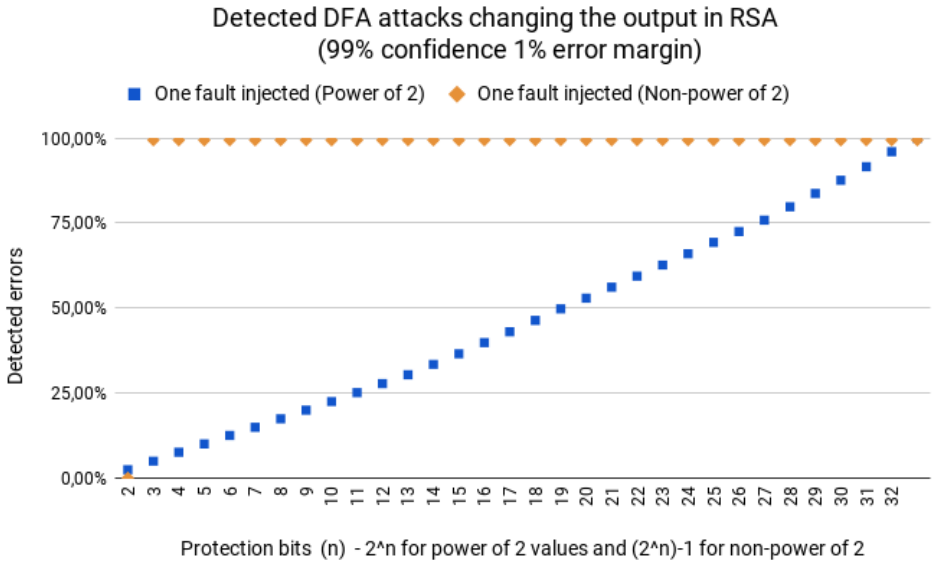


Figure 5.2: Percentage of DFA attacks detection in RSA

Figure 5.3 indicates the results obtained from the evaluation of the protected SHA-1 circuit. This algorithm cannot be protected with non-power-of-two modules, for this reason, the results seem similar to the ones obtained for non-power-of-two values for the RSA algorithm. And consequently, the protection rate is smaller than the one obtained for power-of-two modules in RSA (Figure 5.2). These results were the expected ones as we already knew about this constraint and that the fault detection rate would be reduced due to this limitation.

The evaluation of SHA-1 has been done using the base residue checker and the rotary residue checker defined in subsection 4.1.1. The base residue checker mechanism is represented with green circles while the rotary residue checker is represented with purple triangles in figure 5.3. Comparing the two lines the rotary residue checker gives

better results. In concrete, the rotary residue checker can improve the fault detection rate of the non-modified residue checker up to 84% (for some residue values). Nevertheless, the rotary residue checker has some drawbacks as due to the copies need to do during the rotations it can also decrease the detection rate for some modules down to 4% (for large residue values).

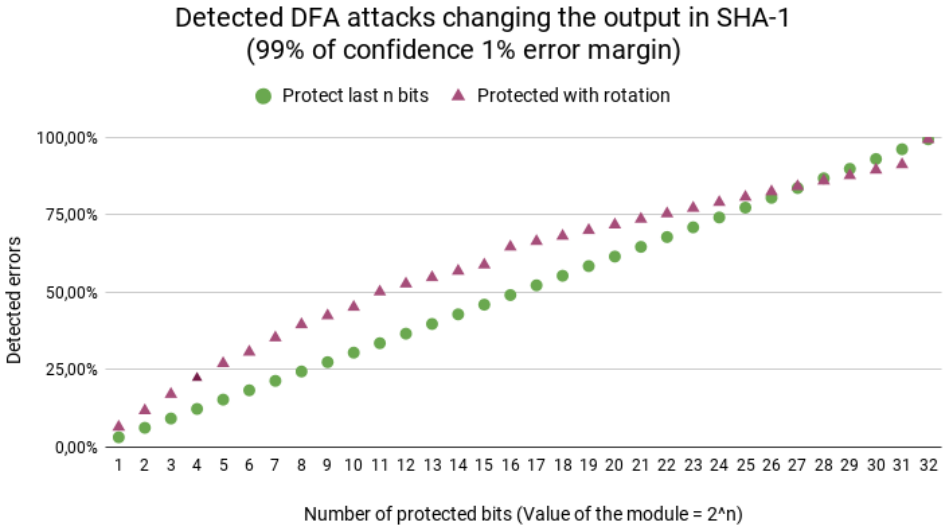


Figure 5.3: Percentage of DFA attacks detection in SHA-1

Figure 5.4 shows the evaluation results of the protection mechanism implemented into the AES algorithm. These results are only shown for non-power-of-two modules as in the case of the SHA-1 algorithm. This is because AES presents the same restriction as SHA-1 of only executing bitwise operations in its algorithm which makes impossible to protect it with non-power-of-two modules using residue checker. For that reason, the results obtained are based on the number of protected bits having a 50% of protection when half of the bits are protected (i.e. value of the module is 2^{16}) and just obtaining the 100% of protection in case the protected signals are fully replicated (i.e. value of the module is 2^{32}).

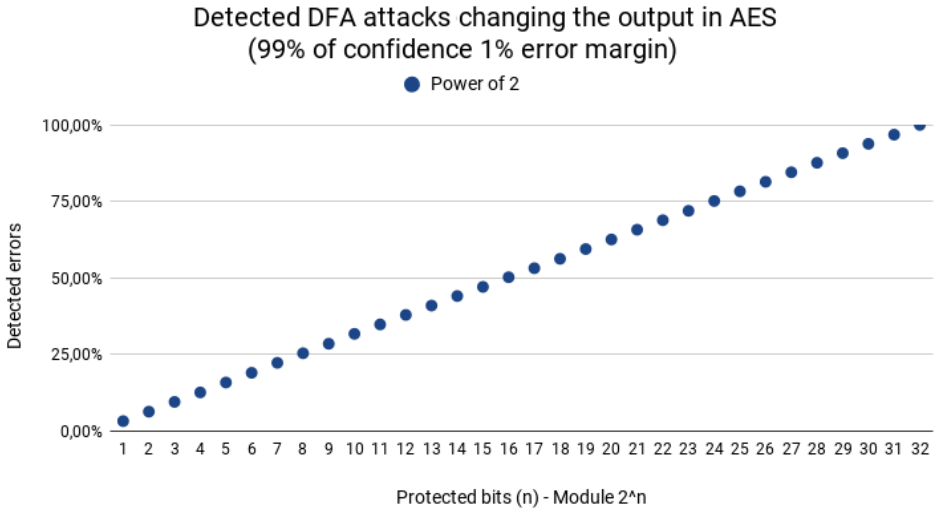


Figure 5.4: Percentage of DFA attacks detection in AES

The rotary residue checker has not been applied to the AES algorithm as the chosen design is composed of component instances that run for a single cycle and then use this output as the input to another component. The signals are protected within these components, for that reason, there is no benefit to the algorithm when applying the rotary mechanism as the protection needs to be recomputed every single cycle. The aim of this proposal is to protect the circuit applying residue checker to the original signals in the circuit but without adding big modifications to the initial non-protected design. Therefore, AES design has not been modified. Still, rotary residue checking could be applied to a sequential version of the AES algorithm, yet, this version of AES is not commonly used due to its poor performance.

Summarising the results for each algorithm, in order to obtain 100% of protection it just needed to protect a few bits using a non-power-of-two module. However, this value of the module cannot be always used as we show for SHA-1 and AES. In these cases, the protected signals have to be fully replicated but as it was described in the development chapter 4 due to the operations performed by the algorithms that are usually based on multiple rounds mixing the values of these signals, it is not needed to fully replicate all the signals in the system so the overhead is smaller than replicating the whole circuit but also getting full protection.

In some DFA attacks, the attacker injects multiple controlled faults, instead of only one per execution, to derive the keys of the algorithm. We, next, evaluate the robustness of our proposed mechanisms against multiple faults (i.e. bit-flips) in the RSA algorithm only. Yet, residue checking should behave in the same way for the other algorithms.

Figure 5.5 shows the behaviour of the residue checker when more errors are applied showing that the protection mechanism can even detect more errors if more faults are injected. In the case of power-of-two modules, injecting two faults increases the residue checker detection by a 19.71% compared to only injecting one fault per simulation. In case of power-of-two modules, the protection is the same, that is why the red line showing the results for two injections is not visible on figure 5.5 as it overlaps with the line representing simulations with one injection.

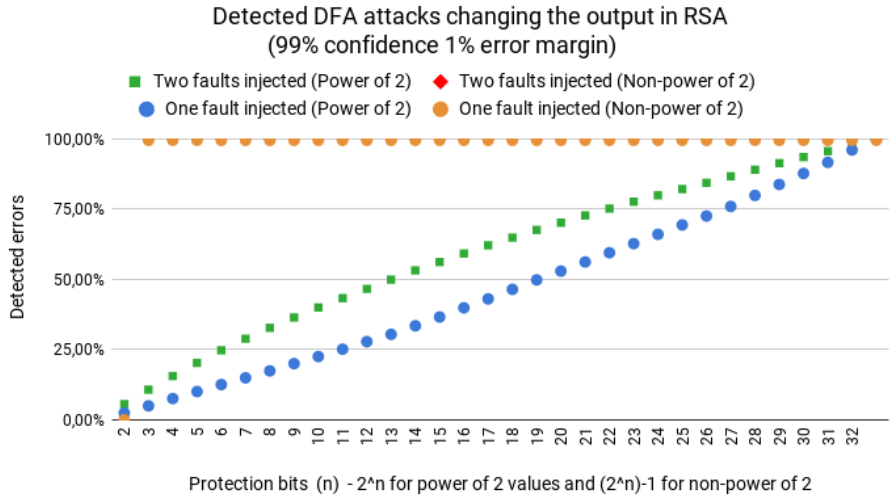


Figure 5.5: Comparison of DFA attacks detection injecting consecutive faults in RSA

5.2.2 Study of the overhead included to the original design

In order to study the overhead that including residue checker has in the systems, the design of each one of the algorithms has been synthesised with and without including the protection mechanism provided by residue checker. Overhead comparisons include the base implementation of each of the algorithm and the largest residue checker simulated. This will give us the worst overhead values.

The synthesis has been applied to two different Xilinx FPGAs belonging to a distinct family. In concrete, the chosen FPGAs are Artix-7[25] and Virtex Ultrascale+[26]. These families are very different between them. Virtex Ultrascale+ is one of the last released products from Xilinx and it provides the highest performance and integration capabilities including many programmable resources compared to other products from Xilinx. Artix-7 is a low-cost board which provides a good performance-per-watt and its number of programmable logic and memory is much more limited than the ones for Virtex Ultrascale+.

Another thing to clarify from the synthesis is the strategy option. Xilinx provides through the Vivado IDE[57] different strategies for the synthesis that can directly synthesise the design as it is or optimise it. There many possible optimisations focusing on the area utilisation, the performance or the runtime. To avoid any possible change on our design we had decided to choose the default synthesis strategy that sets the parameters to the values indicated in listing 5.1.

```
1 // Maximum number of global clock buffers used by synthesis
2 -bugfg = 12
3 // Fanout limit
4 -fanout_limit = 10,000
5 // directive=default implies no optimizations applied
6 -directive = Default
7 // FSM extraction encoding is chosen based on the coding style.
8 -fsm_extraction = auto
9 // Sharing arithmetic operations is chosen based on the coding style.
10 -resource_sharing = auto
11 // control_set_opt_threshold indicates the threshold for synchronous control
12 // set optimizations to lower number of control sets based on the coding style.
13 -control_set_opt_threshold = auto
14 // It indicates the minimum length for chain of registers to be mapped onto SRL
15 -shreg_min_size = 3
16 // max_bram indicates the maximum number of block RAM allowed in the design.
17 // In this case, none.
18 -max_bram = -1
19 // max_uram indicates the maximum number of Ultra RAM allowed in the design.
20 // In this case, none.
21 -max_uram = -1
22 // max_dsp indicates the maximum number of block DSP allowed in the design.
23 // In this case, none.
24 -max_dsp = -1
25 // casca_dsp means to control how adders summing DSP block outputs
26 // will be implemented based on the coding style.
27 -cascade_dsp = auto
```

Listing 5.1: Synthesis strategy parameters

One of the metrics to measure the overhead is the increase in power consumption of the modified design. To calculate this measure, apart form synthesising the design, it is advisable to run a post-synthesis functional simulation of the design in order to obtain information about the circuit and provide a more accurate power report.

Table 5.2 shows the power report obtained for the Virtex Ultrascale+ FPGA. This report includes values for power and temperature but we will focus on the values for the dynamic power as the highlighted column indicates. Dynamic power has been chosen as it is directly related to the design and is not so dependable on the hardware system as the whole power consumption that includes the device static power. From these values, we had calculated the percentage difference between the design including our protection proposal and the original one as in 5.3.

$$Percentage\_different = \frac{|V1 - V2|}{V1} \times 100 \tag{5.3}$$

The result from these calculations is that the power consumption increases by a 5.55% for RSA, by a 4.28% for both SHA-1 modifications and by a 0.88% for AES.

xcvu13p-fhga2104-3-e (Virtex UltraScale+)	**Power consumption(W)**	**Junction temperature(°C)**	**Thermal margin**	**Dynamic power(W)**	**Device static(W)**
RSA	3.084	26.7	73.3 °C (121.8 W)	0.018	3.067
RSA with residue checker	3.085	26.7	73.3 °C (121.8 W)	0.019	3.067
SHA-1	3.209	26.7	73.3 °C (121.7 W)	0.140	3.069
SHA-1 with residue checker	3.216	26.7	73.3 °C (121.7 W)	0.146	3.069
SHA-1 with rotary residue checker	3.216	26.7	73.3 °C (121.7 W)	0.146	3.069
AES	3.413	26.8	73.2 °C (121.5 W)	0.339	3.074
AES with residue checker	3.424	26.8	73.2 °C (121.5 W)	0.342	3.074

Table 5.2: Power consumption synthesising with Virtex UltraScale+

The same power report has been calculated for the Artix-7 FPGA as table 5.3 shows. In this case, the percentage difference in power between the design based on (5.3) are the followings. The protected version of RSA circuit increases its power consumption by a 6.36%, SHA-1 circuit with residue checker consumption rises by an 8% while SHA-1 with rotary residue checker rises by 9.6%, and the modified design of AES consumes a 0.48% more than the non-protected one.

xc7a50tcsg324-1 (Artix 7)	Power consumption(W)	Junction temperature(°C)	Thermal margin	Dynamic power (W)	Device static (W)
RSA	0.081	25.4	59.6 °C (12.4 W)	0.011	0.070
RSA with residue checker	0.081	25.4	59.6 °C (12.4 W)	0.0117	0.070
SHA-1	0.195	25.9	59.1 °C (12.3 W)	0.125	0.070
SHA-1 with residue checker	0.205	26.0	59.0 °C (12.3 W)	0.135	0.070
SHA-1 with rotary residue checker	0.207	26.0	59.0 °C (12.3 W)	0.137	0.070
AES	0.492	27.4	57.6 °C (12 W)	0.415	0.077
AES with residue checker	0.494	27.4	57.6 °C (12 W)	0.417	0.077

Table 5.3: Power consumption synthesising with Artix 7

In general, the consumption obtained indicates that the rise is not more than 0.01 Watts for any case what it is at less than a 10% more consumption compared to the original design. These results also indicate that AES has less overhead in power while SHA-1 has the most. This is because the operation of each cryptographic algorithm are different and implies diverse modifications to protect the design as it is explained in subsections 4.4.3 and 4.3.3 respectively.

Next presented measure is the area utilisation of each design for the two chosen FPGAs. From the are utilisation report, we will focus on the LUTs value to evaluate the area increase of each algorithm.

Table 5.4 shows the resources used for each design using a Virtex Ultrascale+. From these values for the LUTs, column highlighted in table 5.4, we have applied the formula in (5.3) obtaining the following percentage difference between the original and modified design. RSA circuit with residue checker increases a 0.33% in the number of LUTs, SHA-1 circuit increases the number of LUTs in use by a 1.84% and AES circuit does by a 7.81%.

xcvu13p-fhga2104-3-e (Virtex UltraScale+)	CLB LUTs	CLB registers	CARRY8	F7 muxes	F8 muxes	Block RAM Tile	Bounded IOB	Global clock buffer
RSA	596	459	40	-	-	-	132	1
RSA with residue checker	598	459	40	-	-	-	133	1
SHA-1	6881	4254	28	1216	352	-	746	4
SHA-1 with residue checker	7008	4249	28	1216	353	-	763	4
SHA-1 with rotary residue checker	7008	4249	28	1216	353	-	763	4
AES	2560	3968	-	-	-	86	385	1
AES with residue checker	2760	3980	-	-	-	86	400	1

Table 5.4: Area utilisation synthesising with Virtex Ultrascale+

The same calculation in area utilisation has been calculated for Artix-7 FPGA getting these results. RSA circuit rises a 0.33% of LUTs as if the Virtex Ultrascale+ is used, SHA-1 residue checker circuit increases the number of LUTs by a 1.93% while including the rotary residue checker increases SHA-1 by a 1.96%, and the use of LUTs in AES protected circuit goes up by a 7.81%.

xc7a50tcsg324-1 (Artix 7)	**Slice LUTs**	**Slice registers**	**F7 muxes**	**F8 muxes**	**Bonded RAM Tile**	**Bounded IOB**	**BUFGCTRL**
RSA	596	459	-	-	-	132	1
RSA with residue checker	598	459	-	-	-	133	1
SHA-1	6883	4236	1280	352	-	746	4
SHA-1 with residue checker	7016	4241	1280	353	-	763	4
SHA-1 with rotary residue checker	7018	4240	1280	353	-	763	4
AES	2560	3968	-	-	86	385	1
AES with residue checker	2760	3980	-	-	86	400	1

Table 5.5: Area utilisation synthesising with Artix 7

The results for area utilisation seen to be more stable comparing the percentage differences using one FPGA or the other. Unlike the results obtained for power consumption, the design increasing more in area utilisation is AES while RSA is the one with the fewer increment.

Then the maximum frequency of operation for each design has been calculated. These calculations have been done by defining some timing constraints for the design. In particular, the constraints used are the ones in table 5.6 and are taken just as a base value to do the calculation using these same constraints for all the algorithms. These constraint values are not optimised for each one of the design so they should be re-calculated in case the design is going to be implemented.

Signal	**Waveform**	**Period (ns)**	**Frequency (MHz)**
clk	{0.000, 6.400}	12.800	78.125

Table 5.6: Restrictions to calculate the timing constraints of the designs

In order to calculate the maximum frequency, Vivado provides a tool to do a timing analysis based on the predefined timing constraints. However, this report does not directly return the maximum operational frequency but other timing metrics such as the Worst Negative Slack (WNS), Worst Hold Slack (WHS) and Worst Pulse Width Slack (WPWS). So the maximum frequency must be calculated from these reported metrics and the timing constraints set using the following formula 5.4.

$$Fmax = \frac{1}{(Period - WNS)} \tag{5.4}$$

The timing analysis for Virtex Ultrascale+ gives as the following slack values shown in table 5.7. In this table, the maximum frequency has been already calculated and allocate in the first column of this table that it is highlighted.

xcvu13p-fhga2104-3-e (Virtex Ultrascale+)	Max. frequency (MHz)	WNS (ns)	TNS (ns)	WHS (ns)	THS (ns)	WPWS (ns)	TPWS (ns)
RSA	134.390	5.359	0.000	-0.343	-83.527	6.125	0.000
RSA with residue checker	134.390	5.359	0.000	-0.343	-83.527	6.125	0.000
SHA-1	71.921	-1.104	-26.109	-0.933	-486.846	5.932	0.000
SHA-1 with residue checker	74.123	-0.691	-15.103	-3.527	-1087.741	5.932	0.000
SHA-1 with rotary residue checker	73.926	-0.727	-15.710	-3.524	-1099.980	5.932	0.000
AES	136.072	5.451	0.000	-0.311	-295.838	5.905	0.000
AES with residue checker	136.072	5.451	0.000	-0.311	-287.875	5.905	0.000

Table 5.7: Time constraints from synthesis with Virtex Ultrascale+

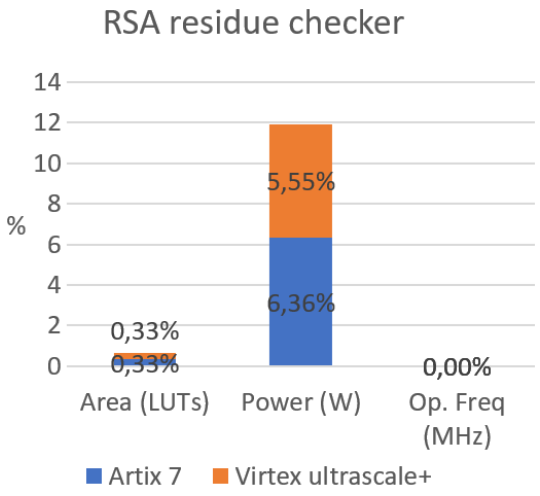
Table 5.8 shows the timing analysis in Artix-7, where the highlighted column contains the maximum frequency values.

xc7a50tcsg324-1 (Artix 7)	Max. frequency (MHz)	WNS (ns)	TNS (ns)	WHS (ns)	THS (ns)	WPWS (ns)	TPWS (ns)
RSA	116.890	4.245	0.000	0.134	0.000	5.900	0.000
RSA with residue checker	116.890	4.245	0.000	0.134	0.000	5.900	0.000
SHA-1	42.938	-10.489	-540.877	-3.791	-762.692	5.150	0.000
SHA-1 with residue checker	44.949	-9.447	-615.120	-3.304	-690.837	5.150	0.000
SHA-1 with rotary residue checker	44.949	-9.447	-615.120	-3.304	-690.837	5.150	0.000
AES	116.890	4.245	0.000	0.139	0.000	5.900	0.000
AES with residue checker	116.890	4.245	0.000	0.139	0.000	5.900	0.000

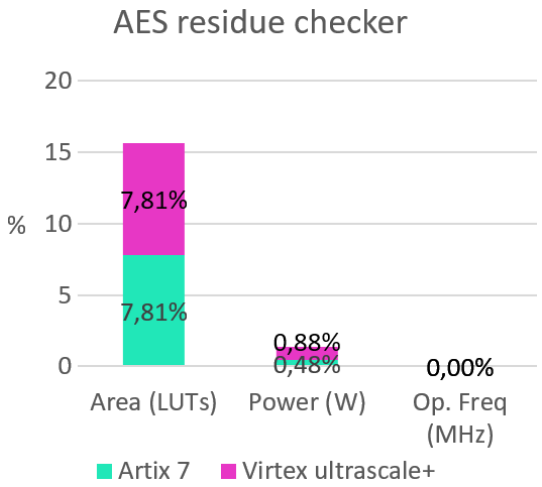
Table 5.8: Time constraints from synthesis with Artix 7

From the maximum frequency values, we can state that our modifications in the design do not change the operational frequency for RSA and AES. But they do for SHA-1 reducing the frequency a 3.06% and 4.68% comparing the original design with the one including residue checker for the Virtex Ultrascale+ and Artix 7, respectively. The operational frequency also gets reduced for the SHA-1 implementing the rotary residue checker a 2.78% for the Virtex Ultrascale+ and 4.68% for the Artix 7.

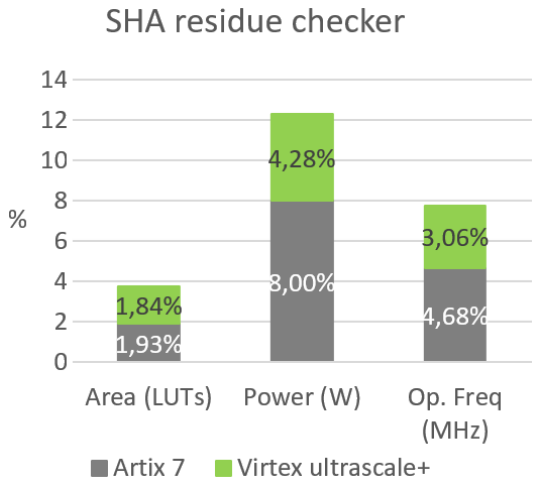
In order to summarise all the results to measure the overhead of the residue checker implementation, the relative values calculated using the percentage difference formula (5.3) has been group on figure 5.6 graph per algorithm showing the obtained values for both of the synthesis with Artix 7 and Virtex Ultrascale+ FPGA.



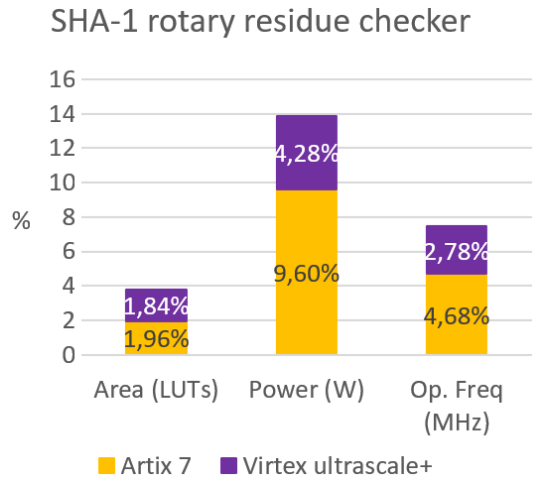
(a) RSA percentage difference between original and modified design



(b) AES percentage difference between original and modified design



(c) SHA-1 percentage difference between original and modified design



(d) SHA-1 percentage difference between original and modified design with rotary residue checker

Figure 5.6: Summary of the overhead because of adding residue checker

Finally, the critical paths of each design have been checked to ensure that none of the changes affects it. The results obtained are positive as the critical path of the designs is not affected by any of the signals included for the residue checker.

6. Publications and future work

The work of this thesis describes a novel method with good results. For that reason, we had decided to publish it for the different attacks and algorithms. Currently, the following publication is already submitted focused on the RSA algorithm, but this work could lead to more publications related to the other cryptographic algorithms.

- Ana Lasheras, Ramon Canal, Eva Rodriguez and Luca Cassano. Protecting RSA Hardware Accelerators against Differential Fault Analysis through Residue Checking. IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, *(Submitted)*, 2019.
- Ana Lasheras, Ramon Canal, Eva Rodriguez and Luca Cassano. Detecting Hardware Trojans in RSA Cores through Residue Checking. IEEE Transactions on Very Large Scale Integration Systems (TVLSI), *(Submitted)*.

6.1 Future work

This proposal is a beginning to test that residue checking can be applied as a security mechanism, but there are other points to continue it like the ones proposed right below.

The evaluation part can be extended to be tested on a physical circuit tested with real DFA attacks[34] and hardware Trojans to obtain more realistic results than the ones given with simulation and synthesis evaluation tools.

This mechanism can be tested on other cryptographic algorithms to strengthen the idea that residue checker can be used as a protection mechanism to security attacks that it started with the test of the three ciphers chosen.

The actual proposal can detect hardware Trojans that manipulate the circuit signals which objective can be to stole sensitive information or Denial of Service (DoS) attacks. However, DOS attacks now can only be detected not corrected so the proposal can be extended for this aim.

Additionally, residue checking can be applied to protect systems against other security attacks manipulating the circuit to get sensitive information.

The current application of residue checker is done following a white box approach where this proposed mechanism is inserted on the design phase of the circuit where all signals are known. But, it can be extended to protect black box IPs.

7. Conclusions

This thesis proposed an innovative mechanism to detect security attacks to cryptographic algorithms using residue checker. Security attacks to computing devices are increasing including hardware attacks, that are the objective of this thesis. In concrete, hardware Trojans and DFA attacks are two of the most common attacks currently used. Three certified cryptographic algorithms have been chosen to implement and test the proposal: i) RSA, ii) SHA and iii) AES . Residue checking has been inserted within an RTL design of these cryptographic algorithms. While power-of-two residue values are compatible with any of the algorithms, non-power-of-two residue values are only applicable to RSA. We have extended the residue checking with a rotary mechanism to extend the detection capabilities of residue checking for power-of-two values.

The evaluation results for each algorithm show that residue checker is a valid method to protect these cryptographic algorithms against hardware Trojans and DFA attacks obtaining 100% of detection to hardware Trojan and to DFA attacks when non-power-of-two modules can be used. While the DFA attacks detection is increased up to 84% using rotary residue checker than doing a signal replication when the algorithm cannot use non-power-of-two modules.

The overhead of the residue checker application increases up to 8% of the area utilisation and its power consumption is below 10% for all the tested algorithms. In concrete, RSA has a low increase in area consumption of 0.33% and power consumption rise up a maximum of 6.36%. The SHA-1 algorithm increases the area up to 1.93% for both implementations of residue checker while power increases at most 9.6%. AES has an increase in the area of 7.81% and a maximum increase in power of just 0.88%. The maximum operating frequency has not been affected for RSA and AES only decreasing for SHA-1 by 4.68% maximum.

In conclusion, residue checking has been shown to be an excellent choice for defending against hardware Trojans and DFA attacks. These results indicate that it can take a relevant place in further research in the security area.

Bibliography

[1] Sally Adee. Syrian radar attack. https://spectrum.ieee.org/semiconductors/design/the-hunt-for-the-kill-switch, 2007. [Online; accessed 29-May-2019].

[2] Adobe. Applying a digital signature using the deprecated sha1 algorithm. https://support.microsoft.com/en-us/help/4472027/2019-sha-2-code-signing-support-requirement-for-windows-and-wsus, 2018. [Online; accessed 01-June-2019].

[3] Atieh Amelian and Shahram Etemadi Borujeni. A side-channel analysis for hardware trojan detection based on path delay measurement. *Journal of Circuits, Systems and Computers*, 27(09):1850138, 2018.

[4] Charles Arthur. Cyber-attack concerns raised over boeing 787 chip's *backdoor*. https://www.theguardian.com/technology/2012/may/29/cyber-attack-concerns-boeing-chip, 2012. [Online; accessed 29-May-2019].

[5] Cryptology Group at Centrum Wiskunde & Informatica (CWI). Attack proof of sha-1 algorithm. https://shattered.io/, 2011. [Online; accessed 01-June-2019].

[6] Anubhab Baksi, Shivam Bhasin, Jakub Breier, Mustafa Khairallah, and Thomas Peyrin. Protecting block ciphers against differential fault attacks without re-keying. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 191–194. IEEE, 2018.

[7] Amin Bazzazi, Mohammad Taghi Manzuri Shalmani, and Ali Mohammad Afshin Hemmatyar. Hardware trojan detection based on logical testing. *Journal of Electronic Testing*, 33(4):381–395, 2017.

[8] Christof Beierle, Gregor Leander, Amir Moradi, and Shahram Rasoolzadeh. Craft: Lightweight tweakable block cipher with efficient protection against dfa attacks. *IACR Transactions on Symmetric Cryptology*, 2019(1):5–45, 2019.

[9] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, Xuan Thuy Ngo, and Laurent Sauvage. Hardware trojan horses in cryptographic ip cores. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 15–29. IEEE, 2013.

[10] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. Hardware trojan attacks: threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, 2014.

[11] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual international cryptology conference*, pages 513–525. Springer, 1997.

[12] Jonathan Cruz, Farimah Farahmandi, Alif Ahmed, and Prabhat Mishra. Hardware trojan detection using atpg and model checking. In *2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID)*, pages 91–96. IEEE, 2018.

[13] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on aes. In *International Conference on Applied Cryptography and Network Security*, pages 293–306. Springer, 2003.

[14] Paul Erdos, Carl Pomerance, and Eric Schmutz. Carmichael's lambda function. *Acta Arith*, 58(4):363–385, 1991.

[15] Christophe Giraud and Adrian Thillard. Piret and quisquater's dfa on aes revisited. *IACR Cryptology ePrint Archive*, 2010:440, 2010.

[16] GlobalSign. A glossary of cryptographic algorithms. `https://www.globalsign.com/en/blog/glossary-of-cryptographic-algorithms/`, 2017. [Online; accessed 04-June-2019].

[17] Support group from DAC. Clusters from the computer architecture department at upc (webpage in catalan). `https://www.ac.upc.edu/app/wiki/serveis-tic/Clusters/Users/arvei?highlight=%28arvei%29`, 2019. [Online; accessed 09-June-2019].

[18] Xiaolong Guo, Huifeng Zhu, Yier Jin, and Xuan Zhang. When capacitors attack: Formal method driven design and detection of charge-domain trojans. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1727–1732. IEEE, 2019.

[19] Scott Hareland, Jose Maiz, Mohsen Alavi, Kaizad Mistry, Steve Walsta, and Changhong Dai. Impact of cmos process scaling and soi on the soft error rates of logic processes. In *2001 Symposium on VLSI Technology. Digest of Technical Papers (IEEE Cat. No. 01 CH37184)*, pages 73–74. IEEE, 2001.

[20] Kento Hasegawa, Masao Yanagisawa, and Nozomu Togawa. Trojan-feature extraction at gate-level netlists and its application to hardware-trojan detection using random forest classifier. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2017.

[21] Wei He, Jakub Breier, and Shivam Bhasin. Cheap and cheerful: a low-cost digital sensor for detecting laser fault injection attacks. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 27–46. Springer, 2016.

[22] Fakir Sharif Hossain, Michihiro Shintani, Michiko Inoue, and Alex Orailoglu. Variation-aware hardware trojan detection through power side-channel. In *2018 IEEE International Test Conference (ITC)*, pages 1–10. IEEE, 2018.

[23] Fakir Sharif Hossain, Tomokazu Yoneda, and Michiko Inoue. An effective and sensitive scan segmentation technique for detecting hardware trojan. *IEICE TRANSACTIONS on Information and Systems*, 100(1):130–139, 2017.

[24] Yuanwen Huang, Swarup Bhunia, and Prabhat Mishra. Mers: statistical test generation for side-channel analysis based trojan detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 130–141. ACM, 2016.

[25] Xilinx Inc. Artix-7 product. `https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html`, 2019. [Online; accessed 10-June-2019].

[26] Xilinx Inc. Virtex ultrascale+ product. `https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html`, 2019. [Online; accessed 10-June-2019].

[27] Marc Joye and Michael Tunstall. *Fault analysis in cryptography*, volume 147. Springer, 2012.

[28] Chong Hee Kim and Jean-Jacques Quisquater. New differential fault analysis on aes key schedule: Two faults are enough. In *International Conference on Smart Card Research and Advanced Applications*, pages 48–60. Springer, 2008.

[29] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[30] Subha Koley and Prasun Ghosal. Addressing hardware security challenges in internet of things: Recent trends and possible solutions. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 517–520. IEEE, 2015.

[31] Daniel Lipetz and Eric Schwarz. Self checking in current floating-point units. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 73–76. IEEE, 2011.

[32] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990, 2018.

[33] Rob Mardisalu. 14 most alarming cyber security statistics in 2019. `https://thebestvpn.com/cyber-security-statistics-2019/`, 2019. [Online; accessed 29-May-2019].

[34] Side-Channel Marvels. phoenixaes: a tool to perform differential fault analysis attacks (dfa) against aes. `https://github.com/SideChannelMarvels/JeanGrey/tree/master/phoenixAES`, 2019. [Online; accessed 10-June-2019].

[35] Microsoft. Changing to sha-2 code signing support. `https://helpx.adobe.com/acrobat/kb/SHA1-algorithm-warning-message.html`, 2019. [Online; accessed 01-June-2019].

[36] Paul Kocher Mike Hamburg and Mark E. Marson. Intel ivy bridge random number generator. `https://www.rambus.com/intel-ivy-bridge-random-number-generator/`, 2012. [Online; accessed 29-May-2019].

[37] P Monteiro and TRN Rao. A residue checker for arithmetic and logical operations. In *2nd Fault Tolerant Computing Symposium*, 1972.

[38] Luong N Nguyen, Chia-Lin Cheng, Milos Prvulovic, and Alenka Zajić. Creating a backscattering side channel to enable detection of dormant hardware trojans. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[39] Michael Nicolaidis. Carry checking/parity prediction adders and alus. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(1):121–128, 2003.

[40] Michael Nicolaidis. Design for soft error mitigation. *IEEE Transactions on Device and Materials Reliability*, 5(3):405–418, 2005.

[41] National Science Foundation (NSF). Benchmarks in the domain of hardware security and trust. `www.trust-hub.org/home`. [Online; accessed 01-June-2019].

[42] National Institute of Standards and Technology. Secure hash standard. *Federal Information Processing Standards Publication FIPS PUB 180*, 11 May 1993.

[43] W Wesley Peterson. On checking an adder. *IBM Journal of Research and Development*, 2(2):166–168, 1958.

[44] Stanisław J Piestrak and Piotr Patronik. Fault-tolerant implementation of direct fir filters protected using residue codes. In *2015 Nordic Circuits and Systems Conference (NORCAS): NORCHIP & International Symposium on System-on-Chip (SoC)*, pages 1–4. IEEE, 2015.

[45] Jeyavijayan Rajendran, Arunshankar Muruga Dhandayuthapany, Vivekananda Vedula, and Ramesh Karri. Formal security verification of third party intellectual property cores for information leakage. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 547–552. IEEE, 2016.

[46] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[47] Indrani Roy, Chester Rebeiro, Aritra Hazra, and Swarup Bhunia. Safari: Automatic synthesis of fault-attack resistant block cipher implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[48] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. A novel technique for improving hardware trojan detection and reducing trojan activation time. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(1):112–125, 2011.

[49] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. *arXiv preprint arXiv:1905.05726*, 2019.

[50] Mukherjee Shubu. Analysis on the minimum number of simulations needed for a certain confidence an error margin. *Architecture Design for Soft Errors*, page 149, 2008.

[51] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. *S&P (May 2019)*, 2019.

[52] Joan Daemen Vincent Rijmen. Specification for the advanced encryption standard (aes). *Federal Information Processing Standards Publication 197*, 2001.

[53] Shugang Wei. Residue checker using optimal signed-digit adder tree for error detection of arithmetic circuits. In *TENCON 2014-2014 IEEE Region 10 Conference*, pages 1–6. IEEE, 2014.

[54] Wikipedia. Advanced encryption standard. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard, 2019. [Online; accessed 08-June-2019].

[55] Wikipedia. Sha.1 pseudocode. https://en.wikipedia.org/wiki/SHA-1#SHA-1_pseudocode, 2019. [Online; accessed 24-June-2019].

[56] Brecht Wyseur. White box cryptography. http://www.whiteboxcrypto.com/, 2011. [Online; accessed 01-June-2019].

[57] Xilinx. Xilinx vivado design suite - hlx editions. https://www.xilinx.com/products/design-tools/vivado.html, 2019. [Online; accessed 04-June-2019].

[58] Junghwan Yoon, Yezee Seo, Jaedong Jang, Mingi Cho, JinGoog Kim, HyeonSook Kim, and Taekyoung Kwon. A bitstream reverse engineering tool for fpga hardware trojan detection. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2318–2320. ACM, 2018.