Protecting RSA Hardware Accelerators against Differential Fault Analysis through Residue Checking

Ana Lasheras, Ramon Canal, Eva Rodríguez
Departament d'Arquitectura de Computadors
Universitat Politecnica de Catalunya
Barcelona, Spain
ana.lasheras@est.fib.upc.edu, {rcanal,evar}@ac.upc.edu

Luca Cassano
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milano, Italy
luca.cassano@polimi.it

***Abstract*—Hardware accelerators for cryptographic algorithms are ubiquitously deployed in nowadays consumer and industrial products. Unfortunately, the HW implementations of such algorithms often suffer from vulnerabilities that expose systems to a number of attacks, among which differential fault analysis (DFA). It is therefore crucial to protect cryptographic circuits against such attacks in a cost-effective and power-efficient way. In this paper, we propose a lightweight technique for protecting circuits implementing the RSA algorithm against DFA. The proposed solution borrows residue checking from the traditional fault tolerance and applies it to RSA circuits in order to first detect the occurrence a fault and then to react to the attack by obfuscating the output values. An experimental campaign demonstrated that the proposed solution detects the 100% of the possible fault attacks while leading to a 2.85% area overhead, a 16.67% power consumption increase and with no operating frequency decrease.**

***Index Terms*—Attack Resistance, Cryptographic Hardware Accelerators, Differential Fault Analysis, Hardware Security**

I. Introduction and Related Work

Cryptographyc hardware accelerators are employed in a plethora of consumer products, such as smart phones and smart cards, where security and performance requirements co-exist. Although on the one hand modern cryptographic algorithms proved to be sufficiently robust from the mathematical point of view [1], their implementations often suffer from a number of vulnerabilities. In particular, in the last years several cryptographyc hardware accelerators demonstrated to be prone to a number of attacks, among which side-channel analysis (SCA) [2] and differential fault analysis (DFA) [3], [4]. Indeed, secret information can leak through side channels, such as the time needed to perform an encription/decription or the power consumed during such operations. As a result, systems may be insecure although featuring such security-dedicated modules.

DFA demonstrated to be a very effective attack technique. It relies on i) injecting maliciously erroneous values into the cryptographic core while it is performing a number of encryptions/decryptions, ii) collecting the incorrect outputs of the circuit, and then iii) analysing the collected outputs to infer secret information, such as the encryption key. The advantage of DFA consists in letting the attacker select both the erroneous values to inject and the points of the circuit where to inject them. This dramatically reduces the amount of collected data needed to obtain the bits of the secret key, thus allowing the attacker to achieve its goal in a reduced time. It is therefore crucial to protect cryptographic circuits against such attacks in a cost-effective and power-efficient way.

Several approaches have been proposed in the last decades to protect cryptographic circuits against fault injection-based attacks. One approach relies on making the implementation physically inaccessible through tamper-proof boxes and on-chip sensors as for the case of the high-end crypto-core IBM 4764 [5]. These approach demonstrated to be particularly effective but very expensive, since they rely on not-standard technologies. More cost-effective solutions aim at detecting faults instead of preventing them. Duplication/triplication and error detecting codes have thus been employed: these solutions are much cheaper than the previous ones at the cost of a high area occupation and power consumption and longer execution time. Often, these solutions have to be tailored for the specific algorithm under protection.

In this paper, we focus on the RSA asymmetric encryption algorithm and we propose a novel approach to protect it against DFA. When looking at protection techniques specific for RSA, one set of works are based on the use of random values during the computation can be found [6], [7]. A different family of works exploit the invertibility of RSA to detect faults during decryption by executing encryption and viceversa [8], [9]. Such techniques protect RSA from attacks based on the observation of the erroneous output. On the other hand, they fail against the so-called *safe error attack*, where the attacker does not exploit the erroneous output but the knowledge of whether the output has been affected by the fault or not [10]. Proposals to tackle such attack have been presented in [11]–[13].

In this paper, we propose a novel protection technique against DFA based on residue checking for circuits implementing the RSA algorithm. To the best of our knowledge, this is the first time RSA is extended with residue checking to reduce its vulnerability to DFA attacks. Our proposal is

to pair the *conventional* RSA implementation with a replica of the RSA algorithm (dubbed *modulo RSA*) that is meant to work with a smaller number of input bits. The values fed into the modulo RSA are the results of a modulo operation of the values fed into the conventional RSA (i.e. their residue values). Thanks to the properties of the modulo operation, at the end of the execution of both replicas the result of the modulo operation on the output of the conventional RSA will match the output of the modulo RSA only if no faults occurred during the processing. On the other hand, this check will fail in case a fault occurred during the processing, thus allowing to identify a possible fault attack. Once a fault attack has been detected, the circuit produces a random output value, thus making a differential fault analysis of the collected output values impossible. The proposed countermeasure allows to detect the 100% of the possible fault attacks while leading to a 2.85% area overhead, a 16.67% power consumption increase and a no operating frequency reduction.

The remainder of this paper is organized as follows: Section II briefly describes the basics of the RSA algorithm and then surveys the existing DFA attacks to RSA also highlighting the ones tackled by our proposal; Section III introduces the basics concepts of residue checking and it then presents the details of the proposed protection technique; Section IV reports about a set of experiments carried out to measure the effectiveness of the proposed solution and the introduced overhead; Finally, Section V concludes the paper.

II. The RSA Algorithm and the Existing Attacks

We first briefly present the basics of RSA and then, we survey the available DFA attacks. We do not discuss general attacks not specifically addressing HW implementations.

A. The RSA Algorithm

The *RSA (Rivest-Shamir-Adleman)* algorithm (originally presented in [14]) is one of the first proposed public-key cryptographic (or asymmetric) algorithms. Unlike in symmetric cryptography, where the same secret key is used for encryption and decryption, in public-key cryptography two keys are employed. The *public* key is used for encryption and the *private* key, which is kept secret by the owner, is used for decryption. It is worth mentioning that RSA can also be used for producing trusted message signatures.

RSA operations involve three large positive integers, namely *e*, *d* (the encryption and decryption keys, respectively) and *n* such that for all integers *m* (with $0 \leq m \leq n$) the following equation holds:

$$(m^e)^d \equiv m \bmod n \tag{1}$$

as well as the same exponentiation with reordered exponents:

$$(m^d)^e \equiv m \bmod n \tag{2}$$

For the sake of space and simplicity, we do not mention here how *e*, *d* and *n* are generated. Nevertheless, it is important to mention that n is obtained as the product of two very large randomly chosen prime numbers p and q (that must also be kept secret) and that e and d are obtained starting from n.

Once *e* and *n* have been distributed, they can be used for encryption. Given a plaintext m, the corresponding cyphertext c can be computed as:

$$c \equiv m^e \bmod n \tag{3}$$

The only way to decrypt c is by using the private key associated with the public key that as been used to encrypt the original plaintext. In other words, only the owner of d can decrypt c by computing:

$$c^d \equiv (m^e)^d \equiv m \bmod n \tag{4}$$

In case RSA is used for generating a message signature s the following equation is computed:

$$s \equiv m^d \bmod n \tag{5}$$

The signature s associated with a message m is considered to be valid if and only if:

$$s^e \equiv (m^d)^e \equiv m \bmod n \tag{6}$$

That means that the signature is valid if and only if it has been produced with the right private key (and thus by the right person) and the message has not been modified.

B. Attacks to RSA

Two families of attacks to RSA do exist: the first one aims at either factoring n (and thus discovering the prime numbers p and q) or directly recovering d; the second one tries to decrypt an encrypted message c without knowing d. A detailed discussion on differential fault attacks to RSA (and other cryptographic algorithms) may be found in [15].

As we previously said, the first family of attacks aims at discovering the private key (or the prime numbers from which it is generated) during either a decryption or a signature computation. The very first type of attacks belonging to this family aimed at entirely discovering the secret information with a singole fault injection. The basic idea behind the *Bellcore attack* (proposed in [16]) is to inject a fault during a signature computation in order to perturbate the nominal processing and to induce the system to compute $\widetilde{s}$ instead of s, where s is the expected fault-free signature and $\widetilde{s} = s + \delta$. Once $\widetilde{s}$ and s have been calculated, the attacker can recover one of the secret prime numbers from which the keys are generated (and thus also the secret key) by calculating the greatest common divisor between $(\widetilde{s} - s)$ and n. A similar attack has been proposed in [17] where a fault is injected during the check of the signature and the prime number is recovered by calculating the greatest common divisor between $(\widetilde{s^e} - m)$ and n. The advantage of such attack is not to require the fault-free signature s.

Another type of attacks belonging to the first family aims at leaking one bit of the secret key at a time under the following assumptions: i) the attacker has arbitrary access to the device; ii) the attacker can chose any ciphertext to be fed into the device; iii) there is no limit to the number of fault injection experiments the attacker is allowed to perform (and thus the fault injection is assumed not to be destructive).

The first attack to RSA belonging to this family has been proposed in [18]. The basic idea of this attack is injecting a number of faults during a number of signature computations where each injected fault leaks a single bit of the private key. A fault during signature computation may be injected such that the corrupted output signature is either $\widetilde{s} = s^{d-2^i}$ or $\widetilde{s} = s^{d+2^i}$ (depending on whether the fault caused a 0-to-1 or a 1-to-0 bit flip) where $i \in [0, v-1]$ and v is the number of bits of the private key. Once $\widetilde{s}$ has been computed, the attacker can calculate the i^{th} bit of the secret key as either $m^{2^i} \bmod n = s/\widetilde{s} \bmod n$ or $m^{2^i} \bmod n = \widetilde{s}/s \bmod n$.

The last type of attacks belonging to the first family aims at discovering the secret key without requiring to observe the output of the circuit [10], [19]. These are the so-called *safe error attacks*, where the attacker only exploits the knowledge of whether the output has been affected by the fault or not. Indeed, in case a fault is injected in a register storing one bit of the secret key, the output of the circuit will be the same as the expected output in case the bit of the key was 0 (flipped to 1 by the fault). On the other hand, the output of the circuit will differ from the expected output in case the bit of the key was 1 (flipped to 0 by the fault). Although very effective, this attack has been demonstrated to be impractical [20].

The second family of attacks to RSA aims at decrypting an encrypted message without any knowledge of the private key, by injecting faults during encryption [20]. Similarly to what happens in the Bellcore attack, the obtained faulty cyphertext is either $\widetilde{c} = c^{e-2^i}$ or $\widetilde{c} = c^{e+2^i}$ where, again, c is the expected chphertext, $i \in [0, v-1]$ and v is the number of bits of the public key. Given the correct cyphertext c and the faulty one $\widetilde{c}$, and being able to compute inverses over $\mathbb{Z}$, the attacker can compute $c \cdot \widetilde{c}^{-1} \bmod n$, thus, easily computing the plaintext associated with c.

The protection methodology proposed in this paper represents an effective and lightweight countermeasure against all the available type of fault attacks except for the safe error attack that, as we mentioned, is very unlikely to be deployed in practice.

III. The Proposed Residue Checking-based Countermeasure

This proposal protects the HW implementation of RSA by pairing the *conventional* implementation with a replica (i.e. the *modulo RSA*). The values fed into the modulo RSA are the results of a modulo operation of the values fed into the conventional RSA (i.e. their residue values). Depending on the modulo value, the resulting replica will have fewer input bits. At the end of the execution, a modulo operation is performed on the output of the conventional RSA; the result of this operation will match the output of the modulo RSA only if no faults occurred during the processing. In case a fault perturbated the processing of the conventional RSA, this check will fail, thus allowing to identify a possible fault attack. Finally, when a fault has been detected, a random output value is produced in order to make the differential fault analysis of the circuit's output unfeasible.

In the remainder of this section, we first recall the basic properties and definitions of residue checking and we then discuss how to select the most appropriate modulo value and how to enhance a HW implementation of RSA with the proposed residue checking-based countermeasure.

A. Detecting Faults through Residue Checking

Given two positive integers a and m (where m is dubbed the *modulo* value), we can calculate a as:

$$a = m * q + r \tag{7}$$

where both q and r are positive integers (q is the *quotient* and r is the *residue*). The operation `modulo(a, m)` is defined as the residue of the integer division of a by m (represented as $|a|_m$) and it returns r.

Residue Checking is a lightweight mathematical mechanisms for statically validating the results produced by arithmetic units. It detects changes on the output data based on the calculation of the residue. For instance, an addition can be checked by testing the equality of Equation 8.

$$|a + b|_m = ||a|_m + |b|_m|_m \tag{8}$$

Indeed, if both sides of the equation are equal, it is likely that no error has occurred. On the other hand, the occurrence of a fault during the processing would for sure make the check fail. Similarly, if no error has occurred the mathematical equality will hold. Thus, when used to detect errors, residue checking guarantees that there are no false positives.

Residue checking works as described for any arithmetical operator. As a consequence, it also works for the modular exponentiation that is a key component of the RSA algorithm. On the other hand, it is worth noting that residue checking does not work for logical operators [21].

B. Selection of the modulo value

On of the key aspects of a fault detection mechanism based on residue checking is the selection of the modulo value. The chosen modulo value strongly affects both the fault detection capability and the introduced overhead.

A modulo value that is a power of 2 can be easily implemented as a logical masking operation, thus reducing the introduced overhead. When the modulo value is a power of 2, given an integer a and a modulo value m, the modulo operation can be expressed as:

$$|a|_m = a \,\&\, (2^m - 1) \tag{9}$$

Consequently, the residue value can be expressed in $m-1$ bits (if treated as an unsigned integer when designing the circuit). On the one hand this choice significantly simplifies the implementation while, on the other hand, it makes the fault detection much less effective. For any power of 2 modulo value, the residue checking operation will miss any modification of the upper bits (i.e. the masked bits). Therefore, the smaller the modulo value, the smaller the introduced overhead but also the lower the fault detection capability (as we will see in in Section IV).

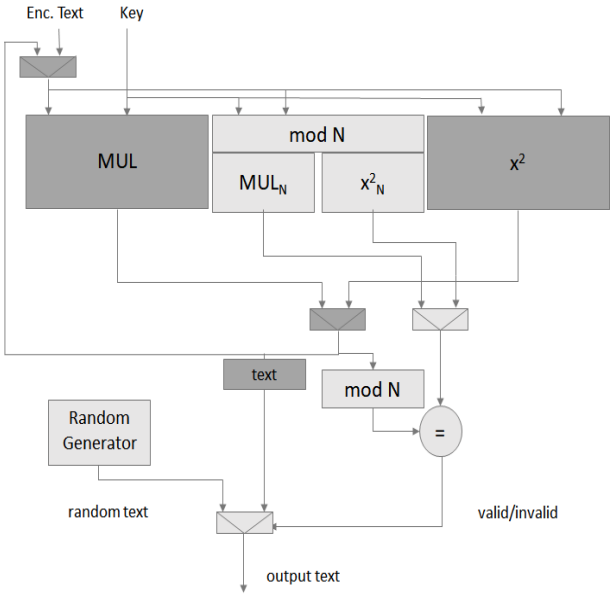

Figure 1. Block Diagram of RSA + Protection Circuit

When the modulo is a non-power of 2 value, the calculation of the residue requires the use of all the bits in a and the fault detection capability does not depend on the chosen modulo value. A naive implementation of the modulo operation for non power of 2 modulo values simply computes the division and then makes a subtraction. Such implementation, although being straightforward, introduces a very high area overhead. Much more optimized implementations with fewer hardware requirements exist [22]. In this line, we rely on the optimized implementation of the modulo operation provided by the Xilinx Vivado™ [23] synthetizer.

C. Implementation of RSA + residue checking

A high-level representation of the RSA hardware implementation enhanced with the proposed protection technique is depicted in Figure 1 where the dark grey blocks belong to the conventional RSA while the light grey ones belong to the modulo RSA. The circuit takes an input text (either plain or encrypted) and a key (either public or private) and produces an output text (either plain or encrypted depending on the requested operation).

The implementation of the conventional RSA is based on the implementation of the modular multiplication operation. As it has been discussed in Section II, the RSA encryption performs the exponent shown in equation (3) and the decryption performs the exponent shown in equation (4). Given two positive integers x and n, x^n (referenced in the text as `Pow(x, n)`) can be computed as the result of the following recursive algorithm:

$$\text{Pow}(x,\, n) = \begin{cases} x, & \text{if } n = 1 \\ \text{Pow}(x^2,\, n/2), & \text{if } n \text{ is even} \\ x \times \text{Pow}(x^2,\, (n-1)/2), & \text{if } n > 2 \text{ is odd} \end{cases}$$

This algorithm is much faster than the ordinary method to perform the exponent calculation. Indeed, when multiplying x by itself, calculating x^n requires n operations. On the other hand, when implementing the algorithm shown above, only $log_2(n)$ operations are needed. The circuit implementing RSA based on such an algorithm for the exponent calculation only counts one multiplier and one square. The feedback loop that brings the output back to the input multiplexer implements the recursivity of the algorithm.

In parallel with the conventional RSA, the modulo RSA takes in input the text and the key. The first step performed by the modulo RSA replica is a modulo operation on both the text and the key. Then the results of such modulo operation are fed into the actual n-bit wide replica of the RSA algorithm where $n = \lceil log_2(m) \rceil$ is the number of bits needed to represent the modulo value m.

The output of the conventional RSA circuit is fed into a modulo operation whose output is compared with the output of the modulo RSA circuit. If these two values are the same, no fault occurred and the output of the conventional RSA circuit is forwarded to the output of the circuit. In case the two modulo values are different, a fault is assumed to have occurred and a random text is forwarded to the output of the circuit in order to make differential fault analysis unfeasible.

IV. Experimental Results

Here, we analyze the effectiveness of the residue checking and the impact on area, power and critical path delay.

A. Experimental Environment

As a real-world HW implementation of the RSA algorithm we considered the trojan-free version of the BASIC RSA-T100 available in the TrustHub repository [24]. Such an RSA HW implementation has then been extended with the proposed residue checking-based protection mechanism. We used Xilinx Vivado™ [23] as a development and synthesis environment and we then employed the accompanying Device Utilization Summary and Power Estimator tools for area, power and time analysis. We considered the Virtex UltraScale+ FPGA (model: xcvu13p-fhga2104-3-e) as a target device.

B. Effectiveness Analysis

The first analysis we carried out aimed at analysing the effectiveness of the proposed countermeasure in detecting fault occurred in the conventional RSA and the impact of the chosen modulo value on the achieved fault detection capability. We implemented a VHDL-level fault simulator that allowed us to emulate the occurrence of faults in both the registers and the output of the conventional RSA circuit. We run a set of fault simulation experiments on a server equipped with one AMD EPYC 7401P (24-Cores) running at 3GHz with 128 GB of DDR4. In each fault simulation experiment, a fault was randomly injected in either a flip-flop or an output bit of the conventional RSA at a random clock cycle.

When dealing with simulation, statistically significant results are usually understood as 99% confidence level at 5%

Table I
NUMBER OF REQUIRED SIMULATIONS PER CONFIDENCE AND ERROR MARGIN VALUE

Confidence Level	99%			95%		
Error Margin	10%	5%	1%	10%	5%	1%
RSA simulations	638	2551	63756	370	1477	36910

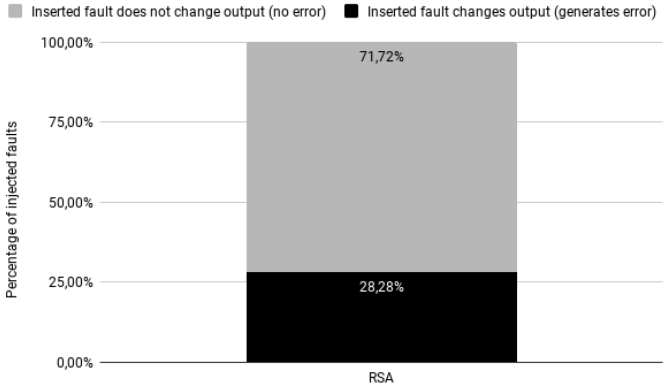

Figure 2. Injected faults and errors caused

error margin. Based on this target, the authors in [25] defined the number of experiments -based on the input signals- needed to reach the selected goal. Table I reports the number of simulations (i.e. different combinations of input values and fault location and clock cycle) needed to achieve the required statistical significance. In this paper, we report the results for 99% confidence and 1% error margin. Consequently, we performed about 64000 simulations for each modulo value (which takes roughly 16 minutes to complete in our server -when using the batch mode of Vivado™).

1) Error Identification: Due to logical and temporal masking, not all faults lead to an erroneous modification of the output (i.e. an error). Moreover, when the circuit is working in the field, this percentage will be increased by electrical masking (this cannot be simulated directly with Vivado™). From a differential fault analysis point of view the *critical* faults are only those that can propagate to the output of the circuit. Figure 2 reports the percentage of faults injected in the conventional RSA that resulted in a modified and unmodified output[1]. The considered HW implementation of the RSA algorithm already tolerates a large percentage of faults (72%). Thus, only the 28% of the faults actually cause an erroneous output of the conventional RSA circuit. These faults (now errors) will be the ones used in the following subsections to assess the effectiveness of the proposed detection technique.

2) Error Detection Capability Analysis: In order to analyze the error detection capability of the proposed technique, we implemented several versions of the protected RSA circuit with several power of 2 and non-power of 2 modulo values. We performed a set of fault injection experiments in which only

[1]We did not inject faults in the modulo RSA because this does not bring any advantage to the DFA attacker. As the residue checking will detect also a mismatch in this case.

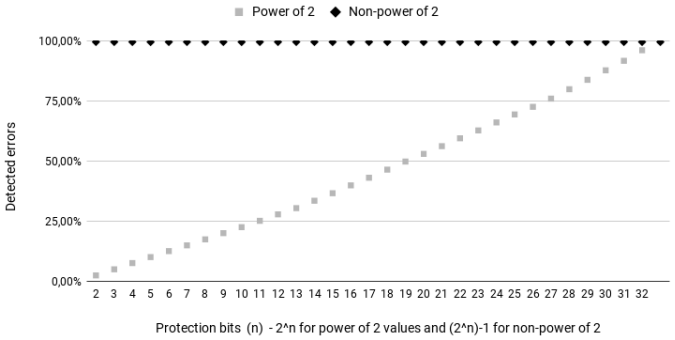

Figure 3. Detection capabilities as a function of the modulo value

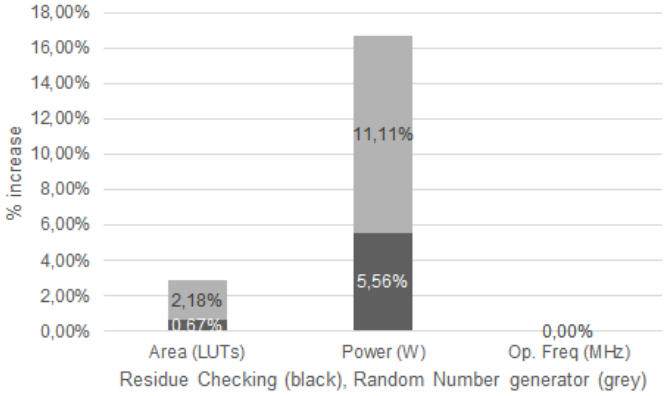

Figure 4. Area and power increase for RSA + Resiude Checking mechanism

the previously identified error generating faults are considered. Figure 3 reports the percentage of detected errors as a function of the number of bits of the modulo value. Given a number of bits (x-axis), two modulo values have been analyzed: 2^n (grey squares) and 2^n-1 (black diamonds). Note that, if the modulo value is a power of 2 then the modulo computation is just a mask operation; otherwise, it requires a full computation with all the bits of the source value (this is actually the advantage of the modulo computation we take advantage of).

Figure 3 clearly shows that any non-power of two modulo value allows to detect all the errors while a conventional power of 2 modulo value is unable to detect all the faults. As we previously mentioned, when a power of 2 modulo value is used, only the lower n bits are stored for checking. Thus, any fault in the upper bits is not detected. Consequently, the detection capabilities of power of 2 modulo values are fundamentally linear to the number of bits considered. When all the bits are considered (i.e. full replication of the circuit), the percentage of errors reaches 100%.

C. Efficiency Analysis

To measure the area, power and critical path delay overhead introduced by the proposed protection technique we synthesized the baseline RSA design and its protected version choosing the $2^{32}-1$ modulo value in order to present the worst-case scenario. Figure 4 reports the results of this analysis.

First, we analysed the area increase. The baseline RSA occupies 596 LUTs, 459 flip-flops, 40 CARRY8 blocks, 132 Bonded IOB and 1 clock buffer. The protected RSA occupies a 2.85% more of LUTs, while all other resources remain the same. This area increase is mostly caused by the random number generator, as shown in the breakdown in Figure 4.

We also measured the power consumption increase by averaging the 64000 executions. The baseline RSA dynamic power consumption is about 180mW while for the protected RSA it is 210mW. The random number generator is always on in order to avoid safe error attacks (as discussed in the introduction). This means the proposal introduces a 16.67% overhead in terms of power. Figure 4 shows also the breakdown between the residue checking computation (5.56%) and the random number generator (11.11%). It is worth mentioning that, in this analysis, we did not consider leakage power since this value would have been affected by all the unused resources in the FPGA and, as a consequence, it cannot be attributed -solely- to the circuit under examination.

Finally, we analyzed the critical path delay increase and thus the operating frequency reduction. Both the baseline RSA circuit and the proposed have a maximum operating frequency of 134,39 MHZ. Consequently, the proposal does not reduce at all the operating frequency. Such a result is explained because the FPGA implementation of both designs is dominated by the path from the input flip-flop, through the multiplier, into the output multiplexor and back to the flip-flop (that stores the partial results). The delay observed in the FPGA is mostly caused by the wiring of this path. By design, also the modulo RSA and the comparison work in parallel with the conventional RSA and thus, the overall effect is not visible in terms of critical path delay.

V. Conclusions

In this paper, we integrate residue checking in a hardware implementation of the RSA cryptographic algorithm to protect against DFA attacks. Our results show that non-power of 2 modulo values provide effective and efficient implementations to detect the occurrence of faults during RSA computation and to trigger the substitution of the erroneous output with a random value. The proposed solution makes differential fault analysis unfeasible.

The presented experimental results demonstrate that the proposed countermeasure allows to detect all the errors with an extremely reduced overhead. Indeed, we measured a worse case 2.85% area increase, a 16.67% dynamic power consumption increase and a no operating frequency slowdown.

References

[1] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[2] Y. Zhou and D. Feng, "Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing." *IACR Cryptology ePrint Archive*, vol. 2005, p. 388, 2005.

[3] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Annual international cryptology conference*. Springer, 1997, pp. 513–525.

[4] M. Joye and M. Tunstall, *Fault analysis in cryptography*. Springer, 2012, vol. 147.

[5] T. W. Arnold, C. Buscaglia, F. Chan, V. Condorelli, J. Dayka, W. Santiago-Fernandez, N. Hadzic, M. D. Hocker, M. Jordan, T. Morris *et al.*, "Ibm 4765 cryptographic coprocessor," *IBM Journal of Research and Development*, vol. 56, no. 1.2, pp. 10–1, 2012.

[6] A. Shamir, "Method and apparatus for protecting public key schemes from timing and fault attacks," Nov. 23 1999, uS Patent 5,991,415.

[7] M. Ciet and M. Joye, "Practical fault countermeasures for chinese remaindering based rsa," in *Workshop on Fault Diagnosis and Tolerance in Cryptography–FDTC*, vol. 5, 2005, pp. 124–132.

[8] M. Joye, "Protecting rsa against fault attacks: The embedding method," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2009, pp. 41–45.

[9] A. Boscher, H. Handschuh, and E. Trichina, "Fault resistant rsa signatures: Chinese remaindering in both directions." *IACR Cryptology ePrint Archive*, vol. 2010, p. 38, 2010.

[10] M. Joye and S.-M. Yen, "The montgomery powering ladder," in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2002, pp. 291–302.

[11] C. Giraud, "An rsa implementation resistant to fault attacks and to simple power analysis," *IEEE Transactions on computers*, vol. 55, no. 9, pp. 1116–1120, 2006.

[12] C. H. Kim and J.-J. Quisquater, "How can we overcome both side channel analysis and fault attacks on rsa-crt?" in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, 2007, pp. 21–29.

[13] S.-M. Yen, S. Kim, S. Lim, and S.-J. Moon, "Rsa speedup with chinese remainder theorem immune against hardware fault cryptanalysis," *IEEE Transactions on computers*, vol. 52, no. 4, pp. 461–472, 2003.

[14] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[15] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[16] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *International conference on the theory and applications of cryptographic techniques*, 1997, pp. 37–51.

[17] A. K. Lenstra, "Memo on rsa signature generation in the presence of faults," Tech. Rep., 1996.

[18] F. Bao, R. H. Deng, Y. Han, A. Jeng, A. D. Narasimhalu, and T. Ngair, "Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults," in *International Workshop on Security Protocols*, 1997, pp. 115–124.

[19] S.-M. Yen and M. Joye, "Checking before output may not be enough against fault-based cryptanalysis," *IEEE Transactions on computers*, vol. 49, no. 9, pp. 967–970, 2000.

[20] A. Barenghi, G. Bertoni, E. Parrinello, and G. Pelosi, "Low voltage fault attacks on the rsa cryptosystem," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2009, pp. 23–31.

[21] J. F. Wakerly, "Principles of self-checking processor design and an example," Departments of Electrical Engineering and Computer Science, Stanford University, Tech. Rep. 115, 1975.

[22] J. T. Butler and T. Sasao, "Fast hardware computation of x mod z," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 294–297.

[23] Xilinx. (2018, dec) Xilinx vivado design suite - hlx editions 2018.3. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[24] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor, "Benchmarking of hardware trojans and maliciously affected circuits," *Journal of Hardware and Systems Security*, vol. 1, no. 1, pp. 85–102, Mar 2017.

[25] "Chapter 3 - architectural vulnerability analysis," in *Architecture Design for Soft Errors*, S. Mukherjee, Ed. Burlington: Morgan Kaufmann, 2008, pp. 79 – 120.