IMPLEMENTATION OF RSA CRYPTOSYSTEM IMMUNE TO TIMING ATTACKS

Ítalo Sampaio, Jamile Martins, Mila Maracaba, Jardel Silveira and Helano Castro

Universidade Federal do Ceará, LESC
Campus do PICI S/N, Bloco 723
Fortaleza, CE – Brasil
E-mail: {italo, jamile, mila, jardel, helano}@lesc.ufc.br

ABSTRACT

The RSA cryptosystem is one of the most popular and most efficient known cryptosystems and its safety is strongly related to the difficulty of breaking it through mathematic tools. A way to perform an attack against it is by using side-channel attacks as a timing attack. In this paper, we describe the FPGA implementation of a complete 512-bits RSA cryptosystem that makes use of the Montgomery's algorithm for modular multiplication. We show that the implemented cryptosystem is immune to timing attacks and we compare it with other kinds of implementations. Finally, we analyze the device utilization in different FPGA's by comparing the number of slices occupied by the implementation.

1. INTRODUCTION

With the advent of the internet, especially in applications as e-commerce, electronic conversations and banking transactions, cryptography has become essential to keep the security of the messages. In all of these applications, we face the problem of sending confidential data through an unsafe channel. Cryptography comes as a solution to this kind of problem, so that only the presumed receptor of the message is able to understand its contents.

Among the techniques used for cryptography nowadays, *RSA cryptography* [8] is the most popular [1]. This popularity comes, in part, thanks to the difficulty of breaking this kind of cryptography by using mathematical tools. Thus, attackers have to use alternative methods for trying to break RSA cryptography. These methods are known as *side-channel attacks* and consist on attacking the physical implementation of the system, instead of attacking its mathematical formulation. One commonly used form of side-channel attack is the timing attack.

In the following sections, we describe the architecture and implementation of a RSA cryptosystem based on the Montgomery's modular multiplication algorithm, showing that the system proposed is immune to timing attacks by comparing its performance with other implementations of RSA cryptography.

2. RSA AND MODULAR EXPONENTIATION

The RSA cryptosystem was first proposed in 1978. It is an example of *public key cryptosystem* [7], that is, two keys are used to perform the cryptography: the public, used to encrypt, and the private, used to decrypt. This system makes use of two large prime numbers in the generation of the keys, and its safety lies in the difficulty of factoring those numbers.

The keys are found in the following way: two big prime numbers, **P** and **Q**, are chosen (nowadays, these numbers are up to 1024 bits long). After that, their product, **N**, and the *Euler Totient Function*, $\phi(N) = (P-1)(Q-1)$ are computed. Then, an integer **E** is chosen so that it is both smaller than N and co-prime with $\phi(N)$, that is, $G.D.C.(\phi(N), E) = 1$. A very common value of **E** is the fourth Fermat number, $F_4 = 2^{16} + 1 = 65537$, because this number is big enough to keep the security of RSA cryptography, and, at the same time, small enough to make fast the operation of encrypting. Finally, we compute **D**, an integer for which $E.D \ mod \ \phi(N) = 1$. Thus, we have the public key (E,N) and the private key (D,N). The encryption and decryption are performed through the pair of inverse functions:

$$Message^E \bmod N = Cipher \tag{2.1}$$

$$Cipher^D \bmod N = Message \tag{2.2}$$

By observing equations (2.1) and (2.2), we can see that, to implement a RSA cryptography system, we need to implement a system that performs the operation of modular exponentiation. The only difference between the encryption and decryption is in the key that will be used as exponent. In the first one, the public key, **E**, is used, and in the second, the private key, **D**.

The modular exponentiation has an extremely high computational cost due to the magnitude of the numbers involved. Therefore, it is necessary to break the exponentiation into several modular multiplications, making the operation more efficient. For RSA cryptography, the most used method is known as *square and multiply*, due to the fact that it carries out the modular exponentiation through modular squaring and modular multiplications.

The square and multiply algorithm, described in [2], is shown in figure. We can see that this algorithm consists of a repetition of two operations. The first one is the modular squaring of the partial result. This can be interpreted as a modular multiplication of $\mathbf{a_0}$ by itself. The second one is the modular product of the message and the partial result. Therefore, the modular exponentiation consists basically of a series of modular multiplications, where only the operands of these multiplications change.

```
Modular Exponentiation
  Input: $m, d = (d_{n-1}, \ldots, d_0), N$
  Output: $m^d \bmod N$
    $a_0 \leftarrow m$
    for $i$ from $n-2$ to $0$ do
      $a_0 \leftarrow a_0^2 \bmod N$
      if $d_i = 1$ then $a_0 \leftarrow m \cdot a_0 \bmod N$
    return $a_0$
```

Figure 1 - Modular Exponentiation algorithm

3. ALGORITHMS FOR MODULAR MULTIPLICATION

For real-time encryption, it is important to have fast modular multiplication algorithms, since the performance of PKC depends primarily on efficiency of modular multiplication.

There are two basic approaches to perform the modular multiplication, as described in [5]: *multiply-then-reduce* and *interleaved*.

In the first method, the multiplication is done first and, after this, the module is subtracted from the partial result repeatedly until the result is achieved. This approach is quite popular, because it allows the use of very efficient pre-existent multiplication algorithms, such as the *multiply-and-reduce* and the *Booth's* algorithm [5], adding to these algorithms only the logic for the modular reduction. However, this approach works well only for small numbers. When used with large numbers, such as in PKC, the partial product tends to be very large, comparing to the module, making necessary a large number of subtractions, which makes the calculation of modular multiplication very slow.

In the interleaved approach the reduction step is performed during the multiplication process, as detailed in [5] and [4]. The main advantage of this approach is the computing time gain, since the modulation is made as soon as the result exceeds the module, decreasing the number of reductions needed. The *Montgomery's algorithm* [7] uses this idea. Moreover, it will be shown that this algorithm has an important characteristic to guarantee timing attack immunity: the response time of the modular multiplier does not depend on the values of the operands. That is why the Montgomery's multiplier was chosen for this work. The algorithm for the Montgomery's modular multiplication is described in [5].

4. TIMMING ATTACKS

In the RSA cryptography, there is only one method to mathematically obtain the private key from the public key. This method consists in factoring the modulus **N**, obtaining the two prime numbers **P** and **Q** [3]. If an attacker manages to obtain the two primes, all he has to do is to reproduce the process to generate the private key from the public key. The problem with this method is the fact that it is not always possible to factor the large number **N**. The capacity of factoring **N** is limited by its size, and whenever computational power increases in a way that this factoring becomes possible, the size of **N** can be increased, thus making impossible to break the cryptosystem.

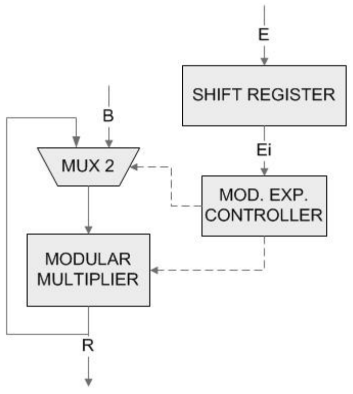

Figure 2 - The Modular Exponentiator with a generic modular multiplier

For this reason, the only way to perform an attack against RSA cryptosystem is by using the side-channel attacks. In the specific case of timing attack, the attacker makes the victim to perform several private key operations, so that he knows the computation time for each operation. The information about the private key is obtained by measuring the variations of the time response. The detailed implementation of the timing attack is not the focus of this work and is described in [6].

From the discussion above, we can conclude that one possible way to make the system immune to timing attacks is to eliminate these variations. In the next section, we will show that the system implemented in this work presents no variation in the response time for a fixed key, what leads to the conclusion that the system is immune against timing attacks.

5. IMPLEMENTATION AND RESULTS

5.1. Architecture

It was already commented in previous sections that, for implementing the modular exponentiation, we need basically a modular multiplier and a control block, which selects whether the operations are performed and also which are the correct operands for each operation. Figure 2 shows the architecture of the modular exponentiator using a generic modular multiplier block, which was implemented using the Add-and-Shift, Booth's and Montgomery's algorithms, in order to compare the results obtained in each implementation.

It is important to note that the Modular Exponentiator Controller block must have control over the Modular Multiplier, selecting the right inputs for each multiplication and deciding whether there must be or not a new modular multiplication. In addition, the controller must make the whole system to wait while the result of a modular multiplication is computed. Thus, the controller is implemented as a Finite State Machine. The states diagram is shown in figure 3 and the complete architecture for the Modular Exponentiation using the Montgomery's modular multiplier is shown in figure 4.

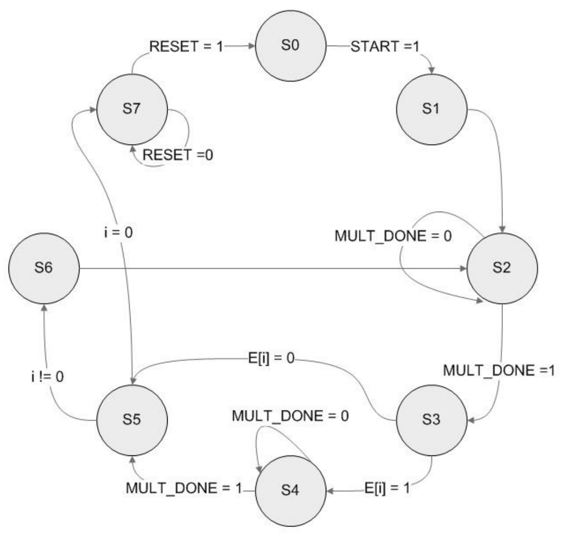

Figure 3 - State Diagram of the controller

The states of the finite states machine are described below:

S0: Clear the inner registers;

S1: Save the value of the input **B** to the register **temp** and load the index **i** with the position of the most significant '1' bit;

S2: Compute the modular product (**temp*temp**) mod **N** and save the result to **temp**;

S3: Check if the bit **E[i]** is equal to '1';

S4: Compute the modular product (**B*temp**) mod **N** and save the result to **temp**;

S5: Check if the index **i** is equal to '0';

S6: Decrement the index **i**;

S7: Load to the output **R** the final result that is in the register **temp**;

5.2. Implementation

This section describes the process of simulation and synthesis of the architectures described in the previous section. The only significant difference among the three implementations is in the modular multiplication block. The design of the three distinct modular multipliers was made by using VHDL and Verilog, and it was based on the implementation described in [4].

The RSA encryption and decryption blocks were described in Systemverilog, by instantiating the modular multiplier blocks, described in their native languages (VHDL and Verilog).

For each algorithm, we built two versions of the RSA cryptosystem module, with different bit lengths: one with 16 bits and the other with 512 bits.

Although the 512-bits implementation is not suitable for practical use [RSA contest], it is a reasonable model for the actual implementations, which use operands of up to 1024 bits. The 16-bits versions are not suitable for practical use at all, but it was necessary to build such versions because the Add-and-Shift and Booth's algorithms make the encryption and decryption operations very slow due to the fact that these algorithms were

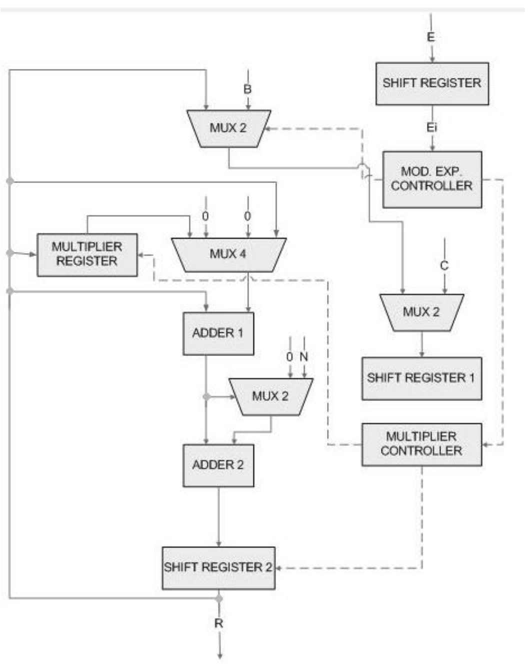

Figure 4 - Complete architecture of the Montgomery Modular Exponentiator

implemented by using the straightforward approach for the modular multiplication. As in the context of our work we are only interested in comparing the time response variations of the three implementations when submitted to different inputs, only the 16-bits versions will be used to this purpose, although we will show the results for the 512-bits implementation of the Montgomery's algorithm.

For the simulation, we used Cadence NC-Sim software and for synthesis, Xilinx ISE 9.2.

The 512-bits RSA cryptosystem using Montgomery's multiplier was implemented on three different FPGA's from the Xilinx family, namely Spartan 3E, Virtex 4 and Virtex II Pro. The results are shown in the next section.

5.3. Results and its analysis

To perform the time tests, we used the 16-bits encryption and decryption modules with the Add-and-Shift, Booth's and Montgomery's modules performing the encryption of words with different sizes, and we measured the time to perform each operation. Table 1 shows the timing results obtained for each algorithm with a 100MHz clock. The major objective of this comparison is to verify the variation of the response time for each implementation caused by inputs with different sizes, since this information could be used for a timing attack. Table 2 shows the results obtained for the 512 bits implementation of the Montgomery module.

By observing the results presented in the tables 1 and 2, we can see that the cryptography module that uses the Montgomery's algorithm is the only one that shows no variation in the response time for any input. The other algorithms present a significant variation in the response time. This fact can be exploited for a timing attack, as

Method	Message length	Encryption time (ns)	Decryption time (ns)
Add-and -Shift	2 bits	1.845.105	1.931.425
	4 bits	2.256.905	4.463.565
	11 bits	1.855.625	3.402.435
	14 bits	3.179.245	3.880.775
Booth	2 bits	1.847.985	1.935.085
	4 bits	2.259.785	4.467.225
	11 bits	1.858.505	3.406.095
	14 bits	3.182.125	3.884.435
Montgomery	2 bits	33.985	43.775
	4 bits	33.985	43.775
	11 bits	33.985	43.775
	14 bits	33.985	43.775

Table 1 - Results for the 16-bits implementations with a clock speed of 100MHz.

Message length	Encryption time (ns)	Decryption Time (ns)
2 bits	1.045.830	46.563.160
16 bits	1.045.830	46.563.160
32 bits	1.045.830	46.563.160
160 bits	1.045.830	46.563.160
210 bits	1.045.830	46.563.160

Table 2 - Results for the 512-bits Montgomery implementation with a clock speed of 100MHz.

FPGA / Device	#Avaliable Slices	# Used Slices	Percentage
Spartan 3E / xc3s250e	4,896	3,680	75%
Virtex 4 / xc4vfx12	10,944	3,680	33%
Virtex II Pro / xc2vp30	27,392	3,680	13%

Table 3 – Area results for the 512-bits RSA cryptosystem.

pointed in [6]. In addition, these algorithms present very low speed due to the fact that the reduction step is performed after the multiplication. Since the results of the multiplications tend to be very large, the reduction step demands a great amount of time to be performed, increasing the time necessary to perform the cryptography. On the other hand, the module that uses the Montgomery's algorithm proves to be completely immune to timing attacks, since the response time shows no variation for different inputs. Besides, the module presents very short response times, even for the decryption operations, which are naturally slow.

The area results for the 512-bits RSA cryptosystem using Montgomery's multiplier on the three different FPGA's are shown in table 3. Note that, although the implemented module occupies a great amount of the area available on the Xilinx Spartan 3E, it is absolutely reasonably to implement it on a Virtex II Pro or even on a Virtex 4.

6. CONCLUSIONS AND FURTHER WORK

Based in the results shown before, we conclude that the RSA cryptosystem based in the Montgomery's modular multiplier proposed in this work has total immunity to timing attacks, without loss of efficiency.

The physical implementation also shows good results, since the system was successfully implemented on three different FPGAs, reaching satisfactory area results.

It is also important to note that the 512-bits key is not a limit. We can easily increase the number of bits for 1024 or even higher bit lengths. Thus, the module presented is an efficient and secure solution.

The above factors show that it is reasonably to think about a silicon implementation of the system, although this shall be done only after a detailed analysis of the power consumption of the system.

In the future, we intend to improve the rate between area and performance. Also, we will analyze the module's performance for other types of side-channel attacks, as the fault and power attacks and perform these attacks on the physical implementation of the system.

7. ACKNOWLEDGEMENTS

The authors are grateful to LESC (Laboratório de Engenharia de Sistema de Computação), for the support to this work and also to Xilinx and Cadence for EDA tools support, provided by their University programs.

8. REFERENCES

[1] A. Daly, and W. Marnane, "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic". In Proc. Of the ACM/SIGDA 10th International Symposium on FP-Gas, pp. 40-49, February 2002.

[2] C. Giraud, "An RSA Implementation Resistant to Fault Attacks and to Simple Power Analysis", IEEE transactions on computer, v. 55, nº.9, September 2006

[3] D. Boneh, "Twenty years of attacks on the RSA cryptosystem", Notices Amer. Math. Soc. 46, pp. 203–213, 1999.

[4] D. Viot, R. Aurélio, H.S. Castro, and R. Jardel, "Modular Multiplication Algorithm For PKC". In: Chip in The Pampa, Gramado, RS, Brazil. VIII SForum on Microelectronics, 2008.

[5] N. Nedjah, and L. Mourelle, "A review of modular multiplication methods and respective hardware implementations". Informatics, nº 30, pp.111–130, 2006.

[6] P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems". Advances in Cryptology-Crypto'96, Lecture Notes in Computer Science, 1109, ACM Press, pp 104-113, 1996.

[7] P.L. Montgomery, "Modular Multiplication without Trial Division", Mathematics of Computation, v. 44, n. 170, pp. 519-521, 1985.

[8] R.L. Rivest, A.Shamir, and L.M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", Communications of the ACM, 21, pp. 120-126, 1978.