İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

DIFFERENTIAL POWER ANALYSIS RESISTANT HARDWARE IMPLEMENTATION OF THE RSA CRYPTOSYSTEM

M.Sc. Thesis by

Keklik ALPTEKİN BAYAM, B.Sc.

Department: Computer Engineering

Programme: Computer Engineering

JUNE 2007

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

DIFFERENTIAL POWER ANALYSIS RESISTANT HARDWARE IMPLEMENTATION OF THE RSA CRYPTOSYSTEM

M.Sc. Thesis by

Keklik ALPTEKİN BAYAM, B.Sc.

(504031518)

Date of submission: 7 May 2007

Date of defence examination: 11 June 2007

Supervisor (Chairman): Prof. Dr. M. Bülent ÖRENCİK

Co-Supervisor (Chairman): Assistant Prof. Dr. S. Berna ÖRS YALÇIN

Members of the Examining Committee Assistant Prof. Dr. Turgay ALTILAR

Assistant Prof. Dr. Feza BUZLUCA

Prof. Dr. Ece Olcay GÜNEŞ

JUNE 2007

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

DİFERANSİYEL GÜÇ ANALİZİNE DAYANIKLI RSA KRİPTO SİSTEMİNİN DONANIM İLE GERÇEKLENMESİ

YÜKSEK LİSANS TEZİ

Müh. Keklik ALPTEKİN BAYAM

(504031518)

Tezin Enstitüye Verildiği Tarih : 7 Mayıs 2007
Tezin Savunulduğu Tarih : 11 Haziran 2007

Tez Danışmanı: Prof. Dr. M. Bülent ÖRENCİK
Tez Eş Danışmanı: Yrd. Doç. Dr. S. Berna ÖRS YALÇIN
Diğer Jüri Üyeleri Yrd. Doç. Dr. Turgay ALTILAR
Yrd. Doç. Dr. Feza BUZLUCA
Prof.Dr. Ece Olcay GÜNEŞ

HAZİRAN 2007

ACKNOWLEDGEMENT

First I would like to thank my supervisors Assistant Prof. Dr. Berna Örs and Prof. Dr. Bülent Örencik for their guidance and support during this thesis work.

I also would like to thank my husband, Fidel, for his love and endless support.

May 2007 **Keklik ALPTEKİN BAYAM**

CONTENTS

TABLE LIST

Page No

FIGURE LIST

Page No

ALGORITHM LIST

Page No

DİFERANSİYEL GÜÇ ANALİZİNE DAYANIKLI RSA KRİPTO SİSTEMİNİN DONANIM İLE GERÇEKLENMESİ

ÖZET

Bu çalışmada, RSA kripto sistemi donanımsal olarak gerçeklenmiş ve daha sonra bir yan kanal analizi çeşidi olan Diferansiyel Güç Analizi (DGA) ile yapılacak saldırılara karşı dayanıklı hale getirilmiştir. RSA kripto sisteminde şifreleme ve şifre çözme, M mesaj, E açık anahtar, N sistem parametresi olmak üzere, M^E (mod N) şeklindeki modüler üs alma işlemi ile yapılır. Bu çalışmadaki RSA kripto sisteminde, Xilinx Sahada Programlanabilir Kapı Dizisi (SPKD (FPGA)) donanım olarak kullanılmıştır. Modüler üs alma işlemi, art arda çarpmalar ile yapılır. Bu gerçeklemede kullanılan Montgomery modüler çarpıcı, Elde Saklamalı Toplayıcılar ile gerçeklenmiştir. Donanım gerçeklemelerinde kullanılan Elde Saklamalı Toplayıcılar, 3 adet *k*-bitlik toplananı, 2 adet *k*-bitlik toplam haline düşürerek, uzun sayıların hızlı çalışma frekanslarında toplanabilmasını sağlarlar. RSA şifreleme algoritmasının işlemleri boyunca Elde Saklamalı gösterilim kullanılmıştır. Böylece çarpıcının işlem hacminin yüksek olması hedeflenmiştir. Çarpıcının 512-bit anahtar uzunluğu kullanarak 140,41 Mbit/s işlem hacmi ile çalıştığı görülmüştür. RSA şifreleme veya şifre çözme işleminin, 512-bit anahtar uzunluğu için, Xilinx XC2V1500 üzerinde ortalama 150,5 Kbit/s işlem hacmine sahip olduğu ve 10240 dilim yer kapladığı görülmüştür. Saldırgan, güç tüketim bilgisinden yararlanarak kripto sistemin gizli anahtarını bulabilir. Bu saldırılara Güç Analizi saldırıları denir ve iki türü vardır: Basit Güç Analizi ve Diferansiyel Güç Analizi saldırıları. Basit Güç Analizi saldırıları tek ölçüm ve gözle tanıma ile yapılırken, Diferansiyel Güç Analizi saldırıları, çok sayıda ölçüm ve istatiksel analiz ile yapılır. Güç Analizi saldırıları, CMOS teknolojisinin günümüzdeki yaygın kullanımından doğan, lojik kapılardaki 0→1 geçişindeki güç tüketimini temel alır. Bu tezde gerçekleştirilen ilk RSA devresinin mimarisi, Basit Güç Analizi saldırılarından gizli anahtarın elde edilmesini engellerken, anahtarın Hamming ağırlığının öğrenilmesine veya Diferansiyel Güç Analizi ile anahtarın kendisinin elde edilmesine karşı duramaz. Diferansiyel Güç Analizine karşı durma yöntemleri arasında donanımsal ve algoritmik çözümler bulunmaktadır. Itoh ve diğ. tarafından önerilen Rastgele Tablolu Pencere Yöntemi (RT-WM) algoritması ile RSA şifreleme algoritmasına getirilen değişiklik, algoritmik karşı durma yöntemlerinden biridir ve donanım üzerinde gerçeklenmemiştir. Bu tezde yapılan ikinci gerçeklemede, ilk gerçeklemenin üzerine bu algoritmanın getirdiği değişiklikler uygulanmıştır. 512-bit anahtar uzunluğu, 2-bit pencere genişliği ve 3-bitlik bir rastgele sayı kullanılarak, Xilinx XCV2600E üzerinde ortalama 18,43 Kbit/s işlem hacmine ve 22712 dilim sayısına ulaşılmaktadır. DGA'ya karşı korumasız ve korumalı her iki mimari, mevcut ölçüm düzeneğinde test edilebilir hale gelmeleri için birer kez de XCV1000E üzerinde gerçeklenmiştir. Korumasız gerçeklemede 81,06 MHz saat frekansı, 104,85 Kb/s işlem hacmi ve 4,88 ms toplam üs alma süresi elde edilmiş ve 9037 dilimlik alan kullanılmıştır. Korumalı gerçeklemede ise 66,66 MHz saat frekansı, 84,42 Kb/s işlem hacmi ve 6,06 ms toplam üs alma süresi elde edilmiş; XCV1000E içinde hazır bulunan blok SelectRAM yapısı ile birlikte 10986 dilimlik alan kullanılmıştır. Korumalı gerçekleme, korumasız ile karşılaştırıldığında, toplam sürenin %24,2 arttığı, işlem hacminin

de %19,5 azaldığı görülmektedir. Tüm donanımsal gerçeklemeler VHDL dili kullanılarak yapılmış; fonksiyonel doğrulama için C/C++ dilleri kullanılmıştır.

DIFFERENTIAL POWER ANALYSIS RESISTANT HARDWARE IMPLEMENTATION OF THE RSA CRYPTOSYSTEM

SUMMARY

In this study, RSA cryptosystem was implemented on hardware and afterwards it was modified to be resistant against Differential Power Analysis (DPA) attacks, which are a type of side channel attacks. The encryption and decryption in an RSA cryptosystem is modular exponentiation, M^E (mod N), where M is the message, E is the public key, and N is a system parameter. In this study, Xilinx Field Programmable Gate Array (FPGA) devices have been used as hardware. Modular exponentiation is realized with sequential multiplications. The Montgomery modular multiplier in this implementation has been realized with Carry-Save Adders. Carry-Save Adders, which are used in hardware implementations, ensure that long numbers are added with fast working frequencies, by reducing 3 *k*-bit summands to 2 *k*-bit sums. Carry-Save representation has been used throughout the RSA encryption algorithm. Thus, the throughput of the multiplier is aimed to be high. The multiplier, implemented on XC2V1500 using 512-bit key length, is observed to be working with a throughput of 140,41 Mb/s. RSA encryption or decryption process for 512-bit key length on Xilinx XC2V1500 takes an average of 150,5 Kb/s throughput and occupies an area of 10240 slices. The attacker can find the secret key of the cryptosystem using the power consumption information. This kind of attacks are called Power Analysis attacks and has two types: Simple Power Analysis and Differential Power Analysis attacks. While Simple Power Analysis attacks are performed with a single measurement and visual recognition, Differential Power Analysis attacks are performed with many measurements and statistical analysis. Power Analysis attacks, are based on the power consumption of 0→1 transitions of the logic gates, which results from the presently common usage of CMOS technology. In this thesis, the primarily implemented RSA circuit's architecture prevents the extraction of the secret key using Simple Power Analysis (SPA) attacks, while it cannot prevent the extraction of the Hamming weight of the key or the extraction of the key using Differential Power Analysis attacks. There are hardware and algorithmic solutions among the countermeasures against Differential Power Analysis. The modification to the RSA encryption algorithm that comes with the Randomized Table Window Method (RT-WM) proposed by Itoh et al. is one of the algorithmic countermeasures and has not been implemented on hardware. In the second implementation of this thesis, the changes within this algorithm have been applied over the first implementation. Realized with 512-bit key length, 2-bit window length, and, a 3-bit random number, on Xilinx XCV2600E, it takes an average of 18,43 Kb/s throughput and an area of 22712 slices is achieved. Both the unprotected and the DPA resistant architectures have been implemented also on XCV1000E, in order for them to become testable with the available measurement setup. The unprotected implementation has resulted in 81,06 MHz of clock frequency, 104,85 Kb/s of throughput, and 4,88 ms of total exponentiation time and occupied an area of 9037 slices. The protected implementation resulted in 66,66 MHz of clock frequency, 84,42 Kb/s of throughput, and 6,06 ms of total exponentiation time and occupied an area of 10986 slices together with the use of the built-in block SelectRAM structure inside XCV1000E. When comparing the

protected implementation with the unprotected, it can be seen that the total time has increased by 24,2%, while the throughput has decreased by 19,5%. All hardware implementations were realized using the VHDL language; and C/C++ have been used for functional verification.

1. INTRODUCTION

1.1 Motivation

RSA is a widely used public-key cryptosystem. RSA encryption is a one-way function, which is not possible to reverse without knowing the private key [1]. RSA is realized with large operands, such that the key length and the operands are greater than or equal to 512 bits. The encryption and decryption in an RSA cryptosystem is modular exponentiation: M^E (mod N). Custom implementations in hardware are more appropriate for the RSA cryptosystem in order to be efficient in area and speed [2].

In this study, a hardware architecture of the RSA cryptosystem has been proposed and implemented on Xilinx FPGA families. In this implementation a Montgomery Modular Multiplier [3] with Carry Save Adder [4] based logic and representation has been used to speed up the calculations.

Side-channel attacks [5] are attacks, based on the information that is retrieved from the device, but is neither the plaintext nor the ciphertext. Power Analysis (PA) attacks [5] are a type of passive side-channel attacks. In these attacks, the power consumption of the circuit is measured while the device is performing an encryption or decryption. The private key or information about the private key is retrieved after an analysis. PA attacks have two types: Simple Power Analysis (SPA) attacks and Differential Power Analysis (DPA) [6] attacks. SPA attacks require a single measurement, while DPA attacks require many measurements followed by a statistical analysis to retrieve information about the private key. There are hardware and algorithmic countermeasures against PA attacks. Itoh *et al.* have proposed an algorithmic countermeasure, Randomized Table Window Method (RT-WM), against Differential Power Analysis (DPA) attacks in [7].

The first implementation in this study prevents the extraction of the private key itself, while it cannot prevent the leakage of the Hamming weight information of the private key when Simple Power Analysis (SPA) attack is implemented. The former protection is due to the architectural design of the circuit. However, the implementation is unprotected against DPA attacks. As the second implementation of this study, RT-WM algorithm [7] has been implemented upon the former unprotected implementation.

1.2 Organization of Thesis

This thesis presents a differential power analysis resistant hardware implementation of the RSA cryptosystem.

Chapter 2 presents the basics of cryptographic systems and explains about the main types of cryptosystems.

Chapter 3 explains the mathematical background behind the RSA cryptosystem.

Chapter 4 gives the fundamentals of RSA architecture both algorithmic and hardware based. This section is the basis to the architectural choices in the implementation.

Chapter 5 presents the basics of side-channel attacks and gives detail about power analysis attacks and the countermeasures against them.

Chapter 6 explains the implementation done within this study: first the unprotected implementation of the RSA cryptosystem, and then the DPA resistant implementation.

Chapter 7 is a review of the thesis and the conclusion is given.

2. CRYPTOGRAPHIC SYSTEMS

The word *cryptography* comes from the Greek words *kryptos* meaning hidden and *graphein* meaning writing. Cryptography is the study of hidden writing, or the science of encrypting and decrypting text [8]. The history of cryptography goes back to Egyptians – about 4000 years ago. In the twentieth century it played a crucial role in both of the world wars. The predominant practitioners of the art were people associated with the military, the diplomatic service and government in general. Cryptography was used as a tool to protect national secrets and strategies [9]. There are two types of cryptosystems: symmetric and public key.

2.1 Symmetric Key Cryptosystems

In symmetric key cryptosystems, Alice and Bob secretly share the key using a secure channel. The exposure of the encryption key or the decryption key renders the system insecure [10].

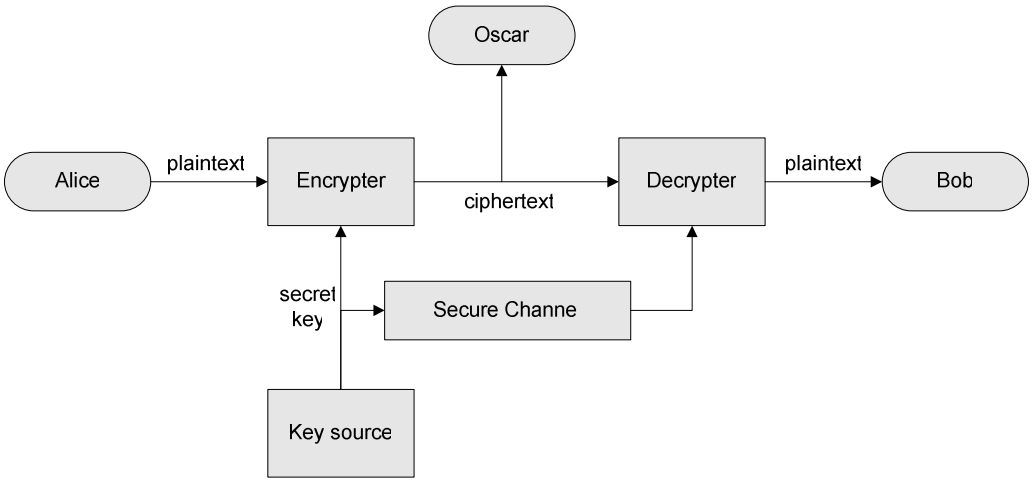

Figure 2.1: Symmetric key cryptosystem communication channel

There are two main problems in symmetric key cryptosystems [10]. The first is the unsafe key exchange. The secure channel between Alice and Bob, which has to be established prior to any communication, might in practice, be very difficult to achieve. Someone can extract the secret key during the key exchange. The second problem is that digital signature is not available in secret key cryptosystems. Since both Alice and Bob share the same secret key, it will be ambiguous who has signed the

plaintext [10]. To overcome these problems, Diffie and Hellman proposed the public key cryptosystems in 1976 [11].

2.2 Public Key Cryptosystems

Diffie and Hellman state in [11] that in a network of n users, where $\left(n^2 - n\right)/2$ pairs can be arranged, it is unrealistic to assume either that all users will be able to wait for a key to be sent by some secure physical means or that keys for all $\left(n^2 - n\right)/2$ pairs can be arranged in advance. They proposed that it was possible to develop systems of the type in which two parties communicating solely over a public channel and using only publicly known techniques can create a secure connection. They had two approaches to the problem, called public key cryptosystems and public key distribution systems.

As proposed by [11], a *public key cryptosystem* is a pair of families $\{E_K\}_{K\in\{K\}}$ and $\{D_K\}_{K\in\{K\}}$ of algorithms representing invertible transformations,

$$E_K : \{M\} \rightarrow \{M\} \quad \textbf{(2.1)}$$

$$D_K : \{M\} \rightarrow \{M\} \quad \textbf{(2.2)}$$

on a finite message space $\{M\}$, such that

1. for every $K \in \{K\}$, E_K is the inverse of D_K,
2. for every $K \in \{K\}$ and $M \in \{M\}$, the algorithms E_K and D_K are easy to compute,
3. for almost every $K \in \{K\}$, each easily computed algorithm equivalent to D_K is computationally infeasible to derive from E_K,
4. for every $K \in \{K\}$, it is feasible to compute inverse pairs E_K and D_K from K.

The third property enables the user to make the encryption algorithm E_K public without compromising the security of his secret decryption algorithm D_K. The cryptographic system now is divided into two as encryption and decryption operations, that given a member of one family of one, it is infeasible to find the corresponding member of the other. The fourth property guarantees that there is a feasible way of computing corresponding pairs of inverse transformations. In practice there must be a true random number generator for generating K, out of which E_K and D_K pair is generated.

With this system, the problem of key distribution is simplified: Each user generates a pair of inverse transformations, E and D and keeps D as secret. The encryption

key E is made public. This means that anyone can encrypt the messages and send them to Bob, while no one else but Bob can decipher the messages intended for him.

In a public key cryptosystem, specifying E specifies a complete algorithm for transforming input messages into output cryptograms. As such a public key system is really a set of trap-door one-way functions, which are not really one-way in that simply computed inverses exist. It is computationally infeasible to find the inverse function out of the forward function. The inverse function can only be easily found with the knowledge of certain trap-door information [11].

In 1977, an public key cryptosystem example, which meets the criteria defined by Diffie and Hellman was proposed by Rivest, Shamir, and Adleman: the RSA cryptosystem [1].

2.3 The RSA Cryptosystem

The RSA cryptosystem [1] uses the same algorithm for both encryption and decryption algorithms. Eq.(2.3) shows the encryption algorithm, where M is the message (plaintext), (E, N) are the public key pair, and C is the ciphertext. Eq.(2.4) shows the decryption algorithm where D is the private key.

$$C = M^{E} \pmod{N} \quad \textbf{(2.3)}$$

$$M = C^{D} \pmod{N} \quad \textbf{(2.4)}$$

The detailed description and the theory behind the RSA algorithm is given in Chapter 3.

3. THE RSA CRYPTOSYSTEM

The RSA cryptosystem was developed by Rivest, Shamir, and Adleman in 1977 [1]. RSA is a public-key cryptosystem that serves both for encryption-decryption and digital signature. Modular encryption is used as encryption and decryption operation in RSA. Modular encryption is a trap-door function, which means that it is easy to compute in one direction, but impossible to calculate its inverse function, which leaves the attacker no choice but to find out the private keys. RSA is used widely in cryptography because of its mathematically strong background.

3.1 Mathematical Background

Let p and q be two distinct large primes, whose product makes up the k-bit modulus N.

$$N = pq,\ p \neq q,\ 2^{k-1} < N < 2^k - 1. \quad \textbf{(3.1)}$$

We select a number E, which will be the public exponent, such that the greatest common divisor of E and $\Phi(N)$ is 1 and E is smaller than N [10],

$$\gcd(E, \Phi(N)) = 1,\ E \in \{1, \cdots, N-1\}, \quad \textbf{(3.2)}$$

where $\Phi(N)$ is Euler's totient function of N given by

$$\Phi(N) = (p-1)\cdot(q-1). \quad \textbf{(3.3)}$$

Afterwards we compute the private key D with

$$D = E^{-1} \bmod(\Phi(N)). \quad \textbf{(3.4)}$$

Usually a small public exponent is selected. The modulus N and E are published, while, D, p, and q are kept secret. RSA encryption is performed by a modular exponentiation operation as shown by Eq.(3.5) where M is the message and C is the ciphertext and $C, M, E \in \{0,1,\cdots,N-1\}$ [1].

$$C = M^E \bmod N,\ C, M, E \in \{0,1,\cdots,N-1\}. \quad \textbf{(3.5)}$$

And RSA decryption is realized through the same function as RSA encryption as shown by Eq.(3.6),

$$M = C^D \bmod N,\ C, M, E \in \{0,1,\cdots,N-1\}, \quad \textbf{(3.6)}$$

where M is the plaintext, C is the ciphertext, N and E are the public keys, and D is the private key. Let us combine Eq.(3.5) and Eq.(3.6):

$$C^D \bmod N = M^{ED} \bmod N. \quad \textbf{(3.7)}$$

Since we have Eq.(3.8)

$$ED = 1 \bmod(\Phi(N)), \quad \textbf{(3.8)}$$

for some integer K, we can write

$$ED = 1 + K\Phi(N). \quad \textbf{(3.9)}$$

When we substitute ED in Eq.(3.7) with Eq.(3.9), we derive Eq.(3.10) and Eq.(3.11) respectively.

$$C^D \bmod N = M^{1+K\Phi(N)} \bmod N, \quad \textbf{(3.10)}$$

$$C^D \bmod N = M \cdot \left(M^{\Phi(N)}\right)^K \bmod N. \quad \textbf{(3.11)}$$

From Euler's theorem we know that, Eq.(3.12) holds for two positive and relatively prime integers a and b

$$a^{\Phi(b)} = 1 \bmod b. \quad \textbf{(3.12)}$$

Using Eq.(3.11) and (3.12), we finally write Eq.(3.13) and (3.14) respectively.

$$C^D \bmod N = M \cdot 1^K \bmod N, \quad \textbf{(3.13)}$$

$$C^D \bmod N = M, \ \gcd(M, N) = 1. \quad \textbf{(3.14)}$$

4. RSA ARCHITECTURE

An RSA encryption is basically a modular exponentiation [1]. When looked with a general perspective, the hardware should include multipliers, adders, dividers, and counters. Even small algorithmic and architectural improvements in the implementation of RSA, which is realized with large operands (> 512 bits), are of big importance. Below are some important points in RSA implementation.

4.1 Exponentiation Methods

The simplest method to realize the modular exponentiation operation $C = M^E \bmod N$, is to start with $C := M \bmod N$ and keep on multiplying the result with M continuously for $E-1$ times [2]. This is obviously the most time consuming and infeasible way to do the exponentiation.

4.1.1 The Binary Method

The "binary method", which is also called the "square and multiply method", scans the bits of exponent E one by one [2]. This scanning can be performed either from left to right or vice a versa. Let E be a k-bit number. The binary method algorithm is given in Algorithm 4.1.

Algorithm 4.1: The Binary Method – left to right

Inputs: $N = (n_{k-1} \cdots n_1 n_0)_2$, $E = (e_{k-1} \cdots e_1 e_0)_2$, $M = (m_{k-1} \cdots m_1 m_0)_2$.
Output: $C = M^E \bmod N$
1. **if** $e_{k-1} = 1$ **then** $C := M$ **else** $C := 1$
2. **for** $i = k-2$ **down to** 0 **do**
3. $C := C \cdot C \bmod N$
4. **if** $e_i = 1$ **then** $C := C \cdot M \bmod N$
5. **return** C

If $e_{k-1} = 1$, the binary method requires $k-1$ squarings and $H(E)-1$ multiplications, where $H(E)$ is the Hamming weight of E. Assuming $E > 0$, which is a must for RSA, this holds for the Hamming weight:

$$0 \le H(E) \le k-1 \qquad \textbf{(4.1)}$$

This gives us an average $H(E)$ of $\frac{1}{2}(k-1)$. The total number of multiplications – assuming the squaring is performed with the same algorithm as multiplication – for the binary method is given in Table 4.1.

Table 4.1: The multiplications required by the binary method

The Binary Method	Multiplications
Maximum	$2(k-1)$
Minimum	$k-1$
Average	$\frac{3}{2}(k-1)$

The number of average multiplications for *k*=512 bit key length is 767.

4.1.2 The *m*-ary Method

The *m*-ary method [12] reduces the number of multiplications processed in an exponentiation. This method is what the binary method would turn into, if we were using *m*-ary representation instead of the binary representation. The exponent E is scanned here r-bits at a time, where $m=2^r$, and $sr=k$. A preprocessing is necessary for the exponentiation process, in which the powers of $M \bmod N$ from 2 to $m-1$ are calculated [2]. This method is more specifically called the "quaternary method" when $m=2$ and the "octal method" when $m=3$. The *m*-ary method is given in Algorithm 4.2.

Algorithm 4.2: The *m*-ary Method

Inputs: $N=(n_{k-1}\cdots n_1 n_0)_2$, $E=(e_{k-1}\cdots e_1 e_0)_2$, $M=(m_{k-1}\cdots m_1 m_0)_2$.

Output: $C=M^E \bmod N$

1. Compute and store $M^w \bmod N$ for $w=2,3,4,\cdots,m-1$
2. Decompose E into r-bit words F_i for $i=0,1,2,\cdots,s-1$, $sr=k$
3. $C \coloneqq M^{F_{s-1}} \bmod N$
4. **for** $i=s-2$ **down to** 0 **do**
5. $\quad C \coloneqq C \cdot C^{2^r} \bmod N$
6. $\quad$ **if** $F_i \neq 0$ **then** $C \coloneqq C \cdot M^{F_i} \bmod N$
7. **return** C

Table 4.2 shows the average number of multiplications (including squarings) required by the *m*-ary method. For the hardware implementation, the m-ary method

requires more area when compared to the binary method; an extra of $m-2$ k-bit registers.

Table 4.2: The average multiplications required by the *m*-ary Method

***m*-ary Method**	**Average multiplications**
Preprocessing	$2^r - 2$
Squarings	$k - r$
Multiplications	$\left(\frac{k}{r} - 1\right)\left(1 - 2^{-r}\right)$
Total	$2^r - 2 + k - r + \left(\frac{k}{r} - 1\right)\left(1 - 2^{-r}\right)$

4.1.3 The Sliding Window Technique

In the *m*-ary method, a zero word makes us skip the multiplication. In order to increase the number of skipped operations and reduce the number of total operations executed, the sliding window technique has been suggested in [12,13]. A sliding window exponentiation algorithm decomposes E into zero and nonzero words, which are called windows. In this technique, nonzero words cannot end with 0. Therefore the multiplications in the preprocessing step are only done to evaluate the odd numbers: $3,5,7,\cdots,\mathrm{m}-1$. The preprocessing multiplications are almost halved.

Two algorithms using this technique are “Constant Length Nonzero Window” (CLNW) proposed by Knuth [12], and “Variable Length Nonzero Window” (VLNW) by Bos and Coster [13]. Both algorithms scan the exponent bits from right to left. In CLNW, the algorithm checks the first bit of the window, if it is a 0, then it becomes a zero window (ZW) and keeps that way until a 1 comes. A 1 starts a nonzero window (NW) and keeps that way for a constant length of *d*-bits. In VLNW algorithm, *d* is the maximum nonzero window length, which means that, during the formation of a NW, we switch to Z when all the remaining bits are all zero. Another variable *q* defines the minimum number of zeros required to switch to ZW. The ZWs are where repetitive squarings are performed, and the NWs require preprocessing at the beginning of the algorithm.

For example, the exponent $E = (111001010001)_2$ is partitioned differently with the mentioned algorithms. The output of CLNW is $E = (\underline{111},00,\underline{101},0,\underline{001})_2$ whilst the output of VLNW is $E = (\underline{111},00,\underline{101},000,\underline{1})_2$.

The analysis performed in [14] shows that the VLNW algorithm requires 5-8% fewer multiplications than the *m*-ary method, namely 6,37% for 512-bit key length.

4.2 Montgomery Multiplication

In 1985 Montgomery introduced a new method for modular multiplication [3]. The approach of Montgomery avoids the time consuming trial division that is a bottleneck for most other algorithms. His method is very efficient and is the basis of many implementations of modular multiplication, both in software and hardware [15].

The modular exponentiation in RSA obviously requires repeated modular multiplications. In 1985, Montgomery introduced an algorithm for computing $R = ab \bmod N$, which is in total, more efficient than first multiplying and afterwards finding the N residue, which would have required k times k-bit additions for the multiplication, and k times k-bit subtractions and comparisons for the division [3]. The Montgomery algorithm computes the result by replacing the division operation with k times the division by a power of 2, where a, b, and n are k-bit binary numbers. Thus, not only computation time, but also area is reduced in hardware implementations. Montgomery multiplication is defined as

$$R' = a'b'r^{-1} \bmod N, \tag{4.2}$$

where $r = 2^k$, and the real multiplicands a and b are needed to be transformed into their N-residues such as

$$a' = a \cdot r \bmod N. \tag{4.3}$$

When Eq.(4.2) and (4.3) are combined, we get

$$R' = arbrr^{-1} \bmod N = abr \bmod N. \tag{4.4}$$

Eq.(4.3) is the preprocessing of Montgomery Multiplication. As R' is not the final result of the multiplication, we need a post-processing, where R' and 1 are the multiplicands of the Montgomery Multiplication, shown in Eq.(4.5).

$$R = (abr) \cdot 1 \cdot r^{-1} \bmod N = ab \bmod N \tag{4.5}$$

The division process is replaced with multiplying by 2^{-k}. Algorithm 4.3 shows how this division is done, which can be realized by simply 1 bit shifting in k steps.

As the processing and preprocessing steps are multiplication processes themselves, the overhead in this multiplication procedure is meaningful only when the

Montgomery Multiplication is done a number of times – for an exponentiation, for example. This makes Montgomery Multiplication suitable for RSA.

In Algorithm 4.3, T_k is inside the interval $(0,2N)$; and therefore a final subtraction is needed if T_k is greater than $N-1$. In [16] this comparison and subtraction operation is omitted by slightly modifying the algorithm. Our implementation uses the Montgomery Multiplication algorithm that has no final subtraction as given in Algorithm 4.4. It saves us from using additional hardware for the comparison and subtraction, by spending two more rounds in the for loop, adding and dividing by 2. Also it will be differential timing attack resistant given in [17]. The operands except the public key N are extended by 1 bit, with a '0' is added as the most significant bit. In Algorithm 4.4, T_k is inside the interval $(0,N)$; and therefore a final subtraction is not needed.

Algorithm 4.3: Montgomery Modular Multiplication with Final Subtraction (*MonPro*)

Inputs: $N=(n_{k-1}\cdots n_1 n_0)_2$, $X=(x_{k-1}\cdots x_1 x_0)_2$, $Y=(y_{k-1}\cdots y_1 y_0)_2$, $r=2^k \bmod N$, $n_0=1$.

Output: $\text{MonPro}(X,Y,N)=XYr^{-1} \bmod N = XY2^{-k} \bmod N$

1. $T_0 := 0$
2. **for** i **from** 0 to $k-1$ **do**
3. **if** $(T_0 + x_i Y)$ is even **then**
4. $T_{i+1} := (T_i + x_i Y)/2$
5. **else** $T_{i+1} := (T_i + x_i Y + N)/2$
6. **if** $T_k \geq N$ **then** $T_k := T_k - N$
7. **return** T_k

Algorithm 4.4: Montgomery Multiplication with No Final Subtraction (*MonPro_NFS*)

Inputs: $N=(n_{k-1}\cdots n_1 n_0)_2$, $X=(x_k\cdots x_1 x_0)_2$, $Y=(y_k\cdots y_1 y_0)_2$, $r=2^{k+2} \bmod N$, $n_0=1$.

Output: $\text{MonPro\_NFS}(X,Y,N)=XYr^{-1} \bmod N = XY2^{-(k+2)} \bmod N$

1. $T_0 := 0$
2. **for** i from 0 to $k+1$ **do**
3. **if** $(T_0 + x_i Y)$ is even **then**
4. $T_{i+1} := (T_i + x_i Y)/2$
5. **else** $T_{i+1} := (T_i + x_i Y + N)/2$
6. **return** T_k

The exponentiation is realized by squaring and multiplications, while the bits of the exponent E are scanned. The number E can be k bits, but it can be less. Therefore the multiplications do not start until the actual most significant bit of E, where the first '1' is seen. Afterwards a squaring is done for every bit of E, and a multiplication is done if the scanned bit is '1'.

When the exponentiation operation uses Montgomery Multiplication Algorithm, it needs a preprocessing, where the N residue of the base number is calculated shown in Eq.(4.3); and a post-processing where the result transferred from the N residue to normal state. A constant number has to be calculated for the preprocessing to evaluate the N residue of the plaintext as shown in Eq.(4.3). This constant number is $2^{2k} \bmod N$ when using *MonPro* algorithm, which becomes $2^{2k+4} \bmod N$ when using *MonPro_NFS*. This constant number can be provided as an input to the function, as it can be calculated directly from the public key N.

Algorithm 4.5: Montgomery Exponentiation with No Final Subtraction (*MonExp_NFS*)

Inputs: $N = (n_{k-1} \cdots n_1 n_0)_2$, $E = (e_{k-1} \cdots e_1 e_0)_2$, $M = (m_{k-1} \cdots m_1 m_0)_2$.

Output: $M^E \bmod N$

1. $Const := 2^{2k+4} \bmod N$
2. $M' := \text{MonPro\_NFS}(M, Const)$
3. $R' := M'$
4. $Start := 0$
5. **for** $i = k-1$ **down to** 0 **do**
6. **if** $Start = 1$ **then**
7. $R' := Mon\text{Pro}\_NFS(R', R')$
8. **if** $e_i = 1$ **then** $R' := Mon\text{Pro}\_NFS(R', M')$
9. **else if** $e_i = 1$ **then** $Start := 1$
10. $R := \text{MonPro\_NFS}(R', 1)$
11. **return** R

4.3 Carry Save Adder

Adders are necessary for the realization of multiplication operations. Adders are necessary for Montgomery multiplication also, namely for step 4 and 5 of Algorithm 4.4. Carry save addition is suitable especially for large operands [4]. It is an appropriate way of reducing 3 k-bit operands to 2 k-bit operands. As a result of this property, Carry Save Adders (CSAs) are used when there are too many inputs to be added, like in the case of multiplication of large operands. CSA has been used in the implemented Montgomery Multiplier within this thesis work. As seen in Figure 4.1, a CSA consists of full adders unconnected with each other. Instead of connecting the carry output of one full adder to the next, like in Carry Ripple Adder, here all carry bits form a line, shifted 1 bit left. The carry input ports are used for the third summand. Thus every time one summand is added to the previous 2 results, a new set of 2 results is formed.

In CSA, there are no horizontal connections, and thus the maximum frequency of the adder is determined by the delay of one full adder, no matter what the size of the

adder is. Thus when a k-bit times k-bit multiplication operation is processed, the result is evaluated at the end of k cycles. CSAs are favorable for Montgomery Multiplication in RSA, where working frequency is important. However it has to be indicated that the result is in carry save representation *(C,S)*. One final addition has to be done to reduce the result from 2 k-bit operands to 1 k-bit operand – to convert back to normal number representation. Carry Ripple Pipelined Adder (CRPA) has been given as an example to this needed adder in the next chapter.

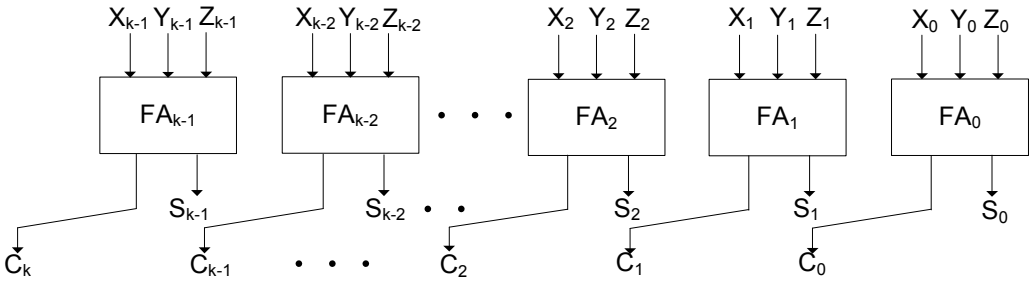

Figure 4.1: Carry Save Adder (CSA)

4.4 Carry Ripple Pipelined Adder

Carry Ripple Adders (CRA) and Carry Look Ahead Adders (CLAA) bring reasonably much delay for large operands [18]. The latter also brings a noteworthy hardware. A CRA of *w*-bit operand size includes w Full Adders (FA) in which the carry output of the *i*th Full Adder is the carry input of the *(i+1)*th Full Adder (Figure 4.2). The delay of the Carry Ripple Adder is the delay of w times the carry delay of one Full Adder, which makes $w(AND + OR + XOR)$ gate delays (See Figure 4.3).

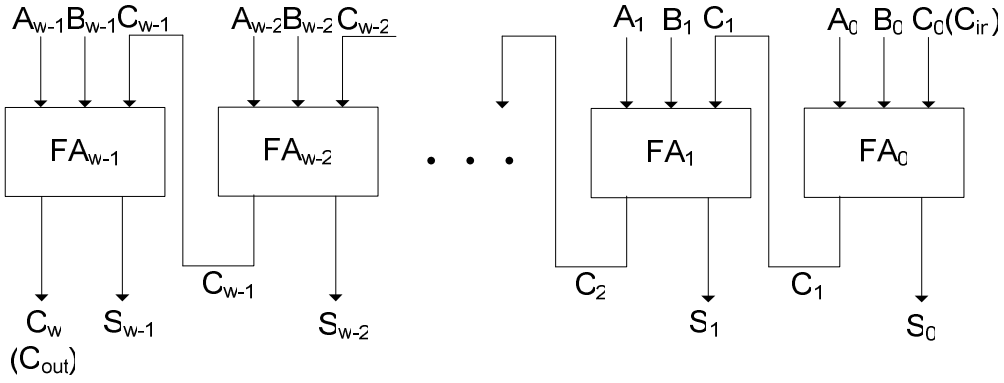

Figure 4.2: Carry Ripple Adder (CRA)

Carry Ripple Pipelined Adder (CRPA) has been used in the implementation of this thesis to add the carry save pair at the end of Montgomery exponentiation and finalize the result. CRPA is a kind of adder constructed by pipelining Carry Ripple Adders (CRA). A CRA of w-bit operand size includes w Full Adders (FA) in which

the carry output of the *i*th Full Adder is the carry input of the *(i+1)*th Full Adder. The delay of the Carry Ripple Adder is the delay of w Full Adders. Therefore it is not suitable for large operands.

The adder to be used with large operands will increase the maximum frequency of the circuit if the execution is done in one clock cycle. Pipelining the addition operation into words is therefore a solution to this problem.

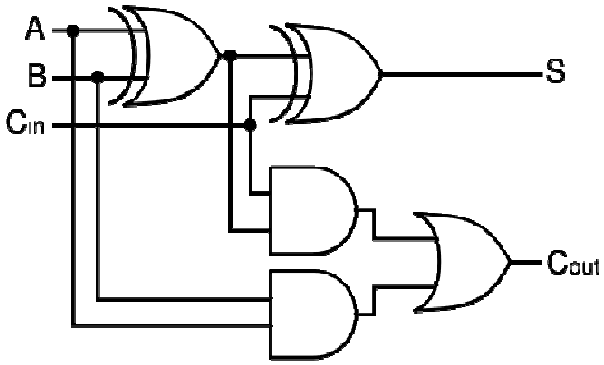

Figure 4.3: Full Adder (FA)

A Carry Ripple Pipelined Adder (CRPA) is a kind of adder constructed by pipelining CRAs. It processes *k*-bit operands word by word by in k/w clock cycles using a w-bit CRAs (Figure 4.4). The carry output of the last FA in the chain, C_w, is registered, and is given to the carry input of the first FA.

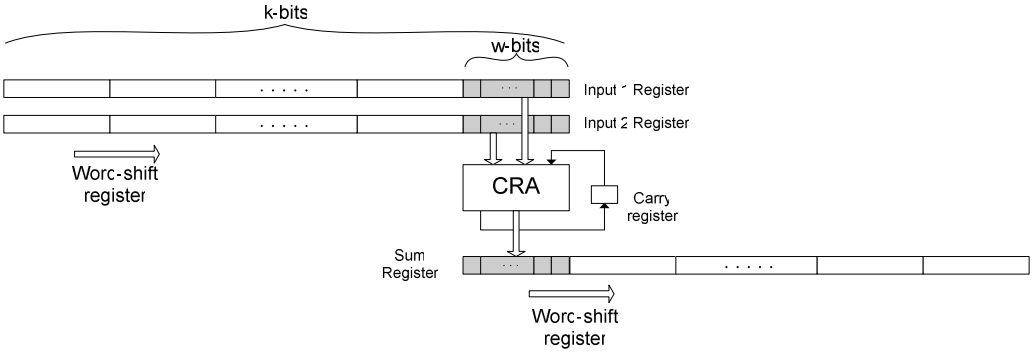

Figure 4.4: Carry Ripple Pipelined Adder (CRPA)

5. SIDE-CHANNEL ATTACKS

In cryptography, an attack based on side channel information is called a “side-channel attack”. Side-channel information is the information that can be retrieved from the encryption device that is neither the plaintext to be encrypted nor the ciphertext resulting from the encryption process [5].

Active attacks, also referred as tampering attacks, require access to the internal circuitry of the attacked device [5]. There are two types:

- Probing attack [19]
- Fault induction attack [20,21]

In passive attacks, the effects of the processing device are measured and used to retrieve the private key. These have mainly four types according to the type of the revealed output:

- Timing Analysis [22]
- Power Analysis [23]
- Electromagnetic Analysis [23]
- Acoustic Analysis [24]

All passive attacks can be either simple or differential. The difference is that, while in simple analysis attacks, the attacker needs only one measurement, he needs numerous measurements and statistics of these measurements in differential analysis attacks.

5.1 Timing Analysis Attacks

For RSA, the square and multiply method is completed with *k* squarings and the number of Hamming weight of the exponent ($H(E)$) multiplications in total. The attacker can calculate the Hamming weight of the exponent by measuring the exponentiation time [22]. One countermeasure to prevent this attack is to always perform a multiplication after each squaring, but not to store the result of the multiplication for the 0 bits. The implementation of this countermeasure gives us a constant of *k* multiplications and *k* squarings, which makes $2k$ multiplications in total.

5.2 Power Analysis Attacks

Power Analysis (PA) attacks are based on analyzing the power consumption of the cryptographic device while it performs encryption or decryption [6]. The physical supporting point of these attacks is that today Complementary Metal Oxide Semiconductor (CMOS) technology is the one to be used most commonly for digital integrated circuit implementations. The power consumption during transitions of a CMOS gate is not the same for $0 \rightarrow 1$ transitions and $1 \rightarrow 0$ transitions. As shown in Figure 5.1, $0 \rightarrow 1$ transitions are using more power than the other. This gives the attacker a good starting point, where he uses Hamming weight information leaks. By this way, the amount of current being discharged can be calculated.

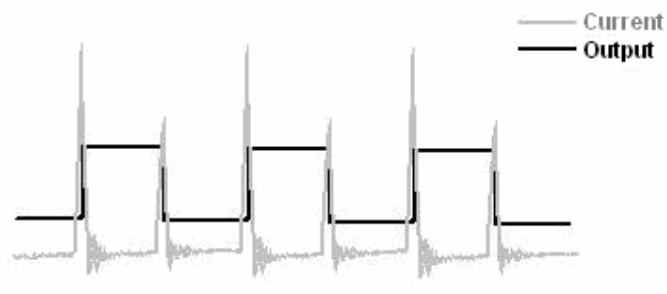

Figure 5.1: The output of a CMOS inverter and the dissipated current

A small (e.g., 50 ohm) resistor inserted in series with the power input of the circuit, in order to measure the change in its power consumption.

5.2.1 Simple Power Analysis Attacks

Simple Power Analysis (SPA) attacks are generally based on looking at the visual representation of the power consumption of a unit while an encryption operation is being performed [6]. SPA is a technique that involves direct interpretation of power consumption measurements collected during cryptographic operations. SPA can yield information about a device's operation as well as key material.

The attacker observes the power consumption of the cryptosystem directly. In RSA, SPA can reveal the difference between multiply and square operations. For this attack to be available on RSA, the system has to either involve a microprocessor, or use different modules for multiplication and squaring if using a Field Programmable Gate Array (FPGA) or an Application Specific Integrated Circuit (ASIC).

5.2.2 Differential Power Analysis Attacks

Differential Power Analysis (DPA) attacks consist not only of visual, but also statistical analysis and error correction statistical methods, to obtain the secret keys

[6]. The attacker monitors the power consumption of the cryptographic device for many inputs, and afterwards analyzes the collected power signal data statistically. Using the result of the statistical analysis, the attacker extracts the secret key. DPA attacks can be used against both secret and private key cryptosystems, stated by Kocher *et al* [6].

There are two types of power consumption leakage that can be observed: the *transition leakage* and the *Hamming weight leakage*. The transition count information leaks when the dominant source of the current is due to switching of the gates. The power dissipated increases with the number of switching gates. The power consumption seen by the measurement from the total power source of a hardware will depend on the total number of gates that switch their states. $0 \rightarrow 1$ transitions have a greater effect than $1 \rightarrow 0$ transitions on the total power consumption [5]. This is taken into account in predictions and mostly, the $1 \rightarrow 0$ transitions are ignored in the calculation.

A Hamming weight leakage occurs when a pre-charged bus design is used. In this case, the number of zeros driven onto the pre-charged bus directly determines the amount of current that is being discharged. This effect can be seen on the falling edges of the output of an inverter. As in the pre-charged bus, if the previous states of the outputs of some gates in the circuit are known and constant for every data, then the power consumption measured from the total power source will give information about the Hamming weight of the current state of these gates [5].

5.3 Countermeasures against Power Analysis Attacks

Countermeasures against PA attacks have two main groups: hardware and software countermeasures [5,25].

5.3.1 Hardware Countermeasures

Hardware countermeasures are usually independent from the encryption or decryption algorithm. They provide a hardware modification to the circuit.

5.3.1.1 Noise Generator

Kocher *et al.* have proposed adding a Random Number Generator (RNG) to increase and randomize the measurement noise [6]. This solution is relatively simple and efficient against attacks, but expensive to implement and not energy efficient. It might be disabled through tampering.

5.3.1.2 Power signal filtering

Coron *et al.* have proposed the power signal filtering method to obscure the measurements [26,27]. While the design might be relatively simple and efficient against attacks, it requires a change to the hardware and might be disabled through tampering. There are two types of filters proposed: active and passive.

5.3.1.3 Novel circuit designs

There are also novel circuit designs which are more specifically targeted to solve the DPA attack problem. Shamir has proposed detachable power supplies [27]. While the design may be relatively simple and efficient against attacks, it may be susceptible to tampering attacks.

5.3.2 Software Countermeasures

Software countermeasures propose an algorithmic solution to the problem.

5.3.2.1 Time randomization

In time randomization method, the order of the operations, or the intervals of operations in an execution are randomized [7,26,28-30]. This method increases the difficulty to attack. It might be cheap to implement in software, however it might be expensive to implement in hardware.

5.3.2.2 Masking techniques

Duplication was proposed by Goubin and Patari in [31] and by Messerges in [32]. This method eliminates the threat of 1st-order DPA, however the device is still susceptible to 2nd-order DPA attacks. Besides, some cryptographic functions may be hard to mask.

5.4 Countermeasures for RSA against Power Analysis Attacks

Throughout this study, the literature has been investigated for countermeasures. Most of the countermeasures for DPA attacks against RSA focus on changing the method of exponentiation from square and multiply to another algorithm that includes some randomness in it. PA countermeasures have some penalties [7]:

- The performance penalty: Especially in exponent splitting, computation time increases. In hardware implementations area can also be a performance penalty [28,30,33].
- Some countermeasures are applicable for RSA, but not all implementations of RSA [29].

- Some countermeasures require additional parameters, such as $\phi(n)$ [26], which belong to the secret key calculation process that is more likely not to be included in the main device.

Walter has proposed in [28] an algorithm called MIST, which generates randomly different addition chains for performing an exponentiation. MIST, making use of a random divisor, makes power attacks which require averaging over a number of exponentiation power traces impossible, and attacks based on recognizing repeated use of the same pre-computed multipliers during an individual exponentiation infeasible. However the algorithm is suited to implementations of software – embedded systems and smart cards. The MIST exponentiation requires a *k*-bit (*k*=key length) divider for the hardware implementation, which gives both the quotient and the remainder as an output. The divider consumes too much area and also time as a result of repeated usage within the proposed algorithm.

In [30], Chevallier-Mames proposes self-randomized algorithms, which use a random number, but also the exponent itself to create randomness. Here an addition chain is created in the preprocessing step. Parts of the exponent are subtracted from itself in each step of the preprocessing. However, the subtracted bits' position, the subtracted range, and the compared parts change in each step. This gives the algorithm too much randomness; which brings security against DPA attacks, whilst it makes it inefficient to be implemented on hardware. The preprocessing time for the hardware implementation also would be infeasible.

The width-w NAF method proposed by Okeya and Takagi in [29] depends on the Nonadjacent Form (NAF) representation stated in [34] by Solinas. The width-*w* NAF method is an efficient window method with small memory, which requires 2^{w-2} points of table. In [29] it is converted to an SPA-resistant addition chain. The proposed construction is optimal in the sense of both efficiency and memory. The memory requirement of scheme is smaller than that of [35], which is based on the signed 2^w-ary method. Unlike the previously explained algorithms, this method does not create only positive members on the addition chain. The NAF representation takes $(32-1)$ instead of 31 for example. Therefore, we could simply say that using NAF representation, the calculation of M^{31} requires 5 squarings plus 1 multiplication with M^{-1} instead of 4 squaring and 4 multiplications. On the other hand, the need for inversion is required for RSA. Modular inversing is an area and time consuming operation, which would be a major offset for the preprocessing. This makes it an infeasible solution for RSA. This method can be feasible for the implementation of Elliptic Curve Cryptography (ECC) [36]. As the squaring corresponds to doubling and multiplication corresponds to addition in ECC, the

division will correspond to a subtraction. This algorithm requires no major extra hardware for the ECC. Also in [35,37,38] similar methods which are infeasible for RSA, but can be feasible for ECC have been used.

Itoh *et al.* in [7] have proposed three algorithms as DPA countermeasures which are applicable to both RSA and ECC cryptosystems. All three countermeasures are based on the window method mentioned earlier in Chapter 4.1.3. In the first algorithm introduced, "Overlapping Window Method" (O-WM), two continuous windows ω_i and ω_{i+1} overlap each other at the same bit position of E, the exponent. Here, ω_i is a random number. An intermediate exponent value is created using ω_i series and the random size of the non overlapped part of the window, h_i. In comparison with the *m*-ary method, the overhead for table making is the same, but the number of repeating the table look-up operations is larger. Besides the processing time penalty of the algorithm, the preprocessing operations have too much randomness, which makes it hard to implement in hardware. The size of the operands in the preprocessing steps is even random.

The second algorithm proposed by Itoh *et al.* [7] is "Randomized Table Window Method" (RT-WM). This algorithm needs a *b*-bit random number r. The exponent is re-calculated using the random number and some intermediate values are formed in return, which are used to form a table. In comparison with the *m*-ary method, the number of repeating table look-up operations are the same, but the overhead for the computation of table-making and normalization are larger.

The third algorithm proposed by Itoh *et al.* [7] is "Hybrid Randomizing Window Method" (HR-WM) is a hybrid technique of the first two, O-WM and RT-WM.

In this study, RT-WM algorithm was implemented as a countermeasure against DPA attacks The RT-WM algorithm is explained in detail in Chapter 5.4.1.

5.4.1 Randomized Table Window Method (RT-WM)

The "Randomized Table Window Method" (RT_WM) algorithm proposed by Itoh *et al.* is given in Algorithm 5.1. This algorithm is a DPA countermeasure both for RSA and ECC based on the window method mentioned earlier in Chapter 4.1.3. The main difference from the window method is that, RT-WM uses randomized data inside the table instead of sequential powers of M.

The subtrahend containing the random number is shifted left in every step by *t-bits* ($t < b$), which creates an overlapping part of $(b-t)$-bits. The subtractions are repeated as long as the result will remain positive. The subtractions result with an intermediate value of Ew which is the concatenation of an array $\omega[i]$ and a normalization value dm.

Algorithm 5.1: RT-WM (Randomized Table Window Method)

Inputs: $N=(n_{k-1}\cdots n_1 n_0)_2$, $E=(e_{k-1}\cdots e_1 e_0)_2$, $M=(m_{k-1}\cdots m_1 m_0)_2$

$Const := 2^{2(k+2)} \bmod N$

Output: $M^E \bmod N$

1. $r :=$ (b-bit random number); /* Generate random number */
2. $\omega\_count := \lceil (k-b)/t \rceil$ /* Pre-computation Phase 1 starts */
3. $subt := r$
4. **for** $i=0$ **to** $\omega\_count-1$ **do**
5. **if** $dw \geq subt$ **then**
6. $dw := dw - E$
7. $subt := subt \cdot 2^t$
8. $dm := (dw_{b-1} dw_{b-2} \cdots dw_1 dw_0)$
9. $\omega_0 := (dw_{k-1} dw_{k-2} \cdots dw_{(\omega\_count-1)\cdot t+b})$
10. **for** $i=1$ **to** $\omega\_count-1$ **do**
11. $\omega_i := (dw_{(\omega\_count-i)\cdot t+b-1} \cdots dw_{(\omega\_count-i-1)\cdot t+b})$
12. $M' = Mon\Pr o\_NFS(M, Const)$ /* Enter MonPro Domain */
13. $Q := M'$ /* Pre-computation Phase 2 starts */
14. $V_0 := M'$
15. **if** $dm=0$ **then**
16. $Q := 0$
17. **for** $i=1$ **to** 2^b-1 **do**
18. $R' := Mon\Pr o\_NFS(R', M')$
19. **if** $i = dm-1$ **then**
20. $Q := R'$
21. **else if** $i = r-1$ **then**
22. $V_o := R'$
23. $U := R'$
24. **for** $i=1$ **to** 2^t-1 **do** /* Pre-computation Phase 3 */
25. $V_i := Mon\Pr o\_NFS(V_{i-1}, U)$
26. $Start := 0$ /* Modular Exponentiation Process */
27. **for** $i=0$ **to** $\omega\_count-1$ **do**
28. **if** $Start=1$ **then**
29. $R' := V_{\omega_i}$
30. **for** j **to** $t-1$ **do**
31. $R' := Mon\Pr o\_NFS(R', R')$
32. **if** $\omega_i \neq 0$ **then**
33. $R' := Mon\Pr o\_NFS(R', V_{\omega_i})$
34. **else if** $\omega_i \neq 0$ **then** $Start := 1$
35. $R' := Mon\Pr o\_NFS(R', Q)$ /* Normalize Data */
36. $R := Mon\Pr o\_NFS(R', 1)$ /* Exit MonPro Domain */
37. **return** R

Figure 5.2 shows the steps in the first part of the pre-processing of the RT-WM algorithm – how the exponent E turns into an array $\omega[i]$ and dm.

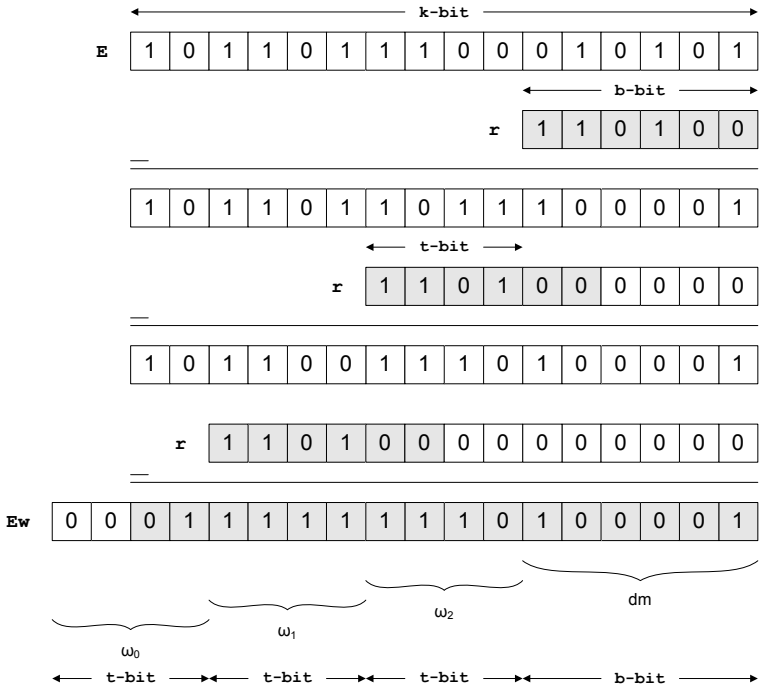

Figure 5.2: Evaluating intermediate values out of the exponent

The recalculation of E determines how the table and the rest of the algorithm works. Eq. 5.1 shows how $\omega[i]$, dm, r, b, and t make up the exponent E.

$$E = \left(\cdots\left(\left(\omega_0 \cdot 2^b + r\right)\cdot 2^t + \omega_1 \cdot 2^b + r\right)\cdot 2^t \cdots\right)\cdot 2^t + \omega_s \cdot 2^b + r + dm \qquad \textbf{(5.1)}$$

The calculation for the table values are given in Eq. 5.2 and computed in pre-computation phases 2 and 3.

$$V_i = M^{\omega_i 2^b + r} \qquad \textbf{(5.2)}$$

Using the values in the table, the rest of the algorithm becomes like “square for 2^t times and multiply with a table value” until the mentioned equation is evaluated. This algorithm brings a preprocessing time, and additional memory for the table is required. An extra subtraction module is not necessary if an adder is already being used within the RSA.

6. IMPLEMENTATION

On the way to achieve a DPA resistant implementation of the RSA cryptosystem, the first step is to implement an unprotected one. The aim of this first step is for the RSA cryptosystem to be functionally correct. The second step is to prove that this implementation cannot stand against DPA attacks. The third step is to choose a countermeasure against DPA attacks and implement upon the unprotected implementation. In this document, the implementations will be called “the unprotected implementation” and “the protected implementation” respectively.

6.1 Unprotected RSA Cryptosystem Implementation

In order to implement the RSA cryptosystem, Montgomery Multiplication block has been realized with MonPro_NFS_CSA algorithm, which is given as Algorithm 6.1. This algorithm does no final subtraction like in the previously explained Algorithm 4.4. When Montgomery multiplication is realized using normal number representation, the operands look like in Figure 6.1. When it is realized using Carry Save representation then the multiplicand, multiplier and the result are doubled as Carry and Save, shown in Figure 6.2.

$$
\begin{array}{rl}
 & Y_k \; Y_{k-1} \cdots Y_1 \; Y_0 \quad \text{multiplicand} \\
x \pmod N & X_k \; X_{k-1} \cdots X_1 \; X_0 \quad \text{multiplier} \\
\hline
 & R_k \; R_{k-1} \cdots R_1 \; R_0 \quad \text{result}
\end{array}
$$

Figure 6.1: Operands of a standard Montgomery multiplier

$$
\begin{array}{rl}
 & \begin{pmatrix} YC_k YC_{k-1} \cdots YC_1 \; YC_0 \\ YS_k YS_{k-1} \cdots YS_1 \; YS_0 \end{pmatrix} \text{multiplicand} \\
 & \begin{pmatrix} XC_k XC_{k-1} \cdots XC_1 \; XC_0 \\ XS_k XS_{k-1} \cdots XS_1 \; XS_0 \end{pmatrix} \text{multiplier} \\
x \pmod N & \\
\hline
 & \begin{pmatrix} RC_k RC_{k-1} \cdots RC_1 \; RC_0 \\ RS_k RS_{k-1} \cdots RS_1 \; RS_0 \end{pmatrix} \text{result}
\end{array}
$$

Figure 6.2: Operands of a Montgomery multiplier using Carry Save Representation

The RSA Encryption/Decryption algorithm, which uses Montgomery Multiplication, also changes accordingly and it is named MonExp_NFS_CSA [39], given in

Algorithm 6.2. The adder required by the encryption process is realized as CRPA, explained in Chapter 4.4.

Algorithm 6.1: Montgomery Multiplication with No Final Subtraction using Carry Save Adder Representation (*MonPro_NFS_CSA*)

Inputs: $XC = (xc_{k+1} \cdots xc_1 xc_0)_2$, $XS = (xs_{k+1} \cdots xs_1 xs_0)_2$,
$YC = (yc_{k+1} \cdots yc_1 yc_0)_2$, $YS = (ys_{k+1} \cdots ys_1 ys_0)_2$, $N = (n_{k-1} \cdots n_1 n_0)_2$,
$r = 2^{k+2} \bmod N$, $n_0 = 1$.

Output: $(RC, RS) = (XC, XS) \cdot (YC, YS) \cdot r^{-1} \bmod N$

1. $TC = (tc_{k+1} \cdots tc_1 tc_0)_2$, $TS = (ts_{k+1} \cdots ts_1 ts_0)_2$
2. $TC_0 := 0$; $TS_0 := 0$
3. **for** i from 0 **to** $k+1$ **do**
4. $x_i := xc_i + xs_i$
5. $(C1_i, S1_i) := TC_i + TS_i + x_i \cdot YC_0$
6. $(C2_i, S2_i) := C1_i + S1_i + x_i \cdot YS_0$
7. **if** $s2_{i0} = 0$ **then**
8. . $(TC_i, TS_i) := (C2_i + S2_i + 0)/2$
9. **else** $(TC_i, TS_i) := (C2_i + S2_i + N)/2$
10. **return** (TC_{k+1}, TS_{k+1})

Algorithm 6.2: RSA Encryption with Montgomery Multiplication with No Final Subtraction using Carry Save Adder Representation (*MonExp_NFS_CSA*)

Inputs: $N = (n_{k-1} \cdots n_1 n_0)_2$, $E = (e_{k-1} \cdots e_1 e_0)_2$, $M = (m_{k-1} \cdots m_1 m_0)_2$
$Const := 2^{2(k+2)} \bmod N$

Output: $M^E \bmod N$

1. $Start := 0$
2. $(MC', MS') := Mon\,Pro\_NFS\_CSA(M, 0, Const, 0, N)$
3. $(RC', RS') := (MC', MS')$
4. **for** $i = k-1$ **down to** 0 **do**
5. **if** $Start = 1$ **then**
6. $(RC', RS') := Mon\,Pro\_NFS\_CSA(RC', RS', RC', RS', N)$
7. **if** $e_i = 1$ **then**
8. $(RC', RS') := Mon\,Pro\_NFS\_CSA(RC', RS', MC', MS', N)$
9. **else if** $e_i = 1$ **then** $Start := 1$
10. $(RC, RS) := Mon\,Pro\_NFS\_CSA(RC', RS', 1, 0, N)$
11. $R := RC + RS$
12. **return** R

Two modules have been used inside the top level module: MonExp_NFS_CSA and a communication module PC2FPGA. Inside MonExp_NFS_CSA there is MonPro_NFS_CSA and CRPA. Inside CRPA, there is a CRA. Figure 6.3 shows the I/O ports, blocks, and connections, and important registers inside the RSA implementation.

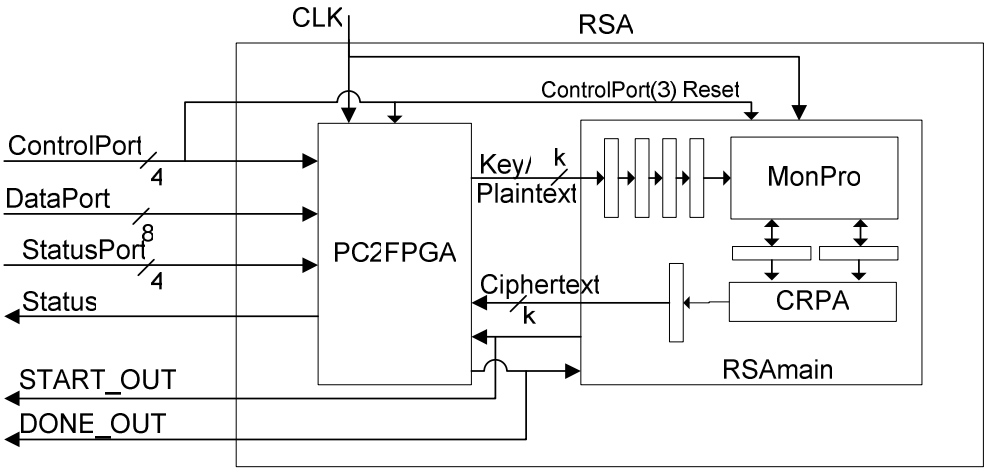

Figure 6.3: RSA module and its blocks

6.1.1 Hardware Implementation

Figure 6.4 shows the main processing element of the hardware implementation using CSA representation, which was functionally described in Algorithm 6.2, *MonPro_NFS_CSA*. There are three levels of CSAs, which determine the multiplier's delay.

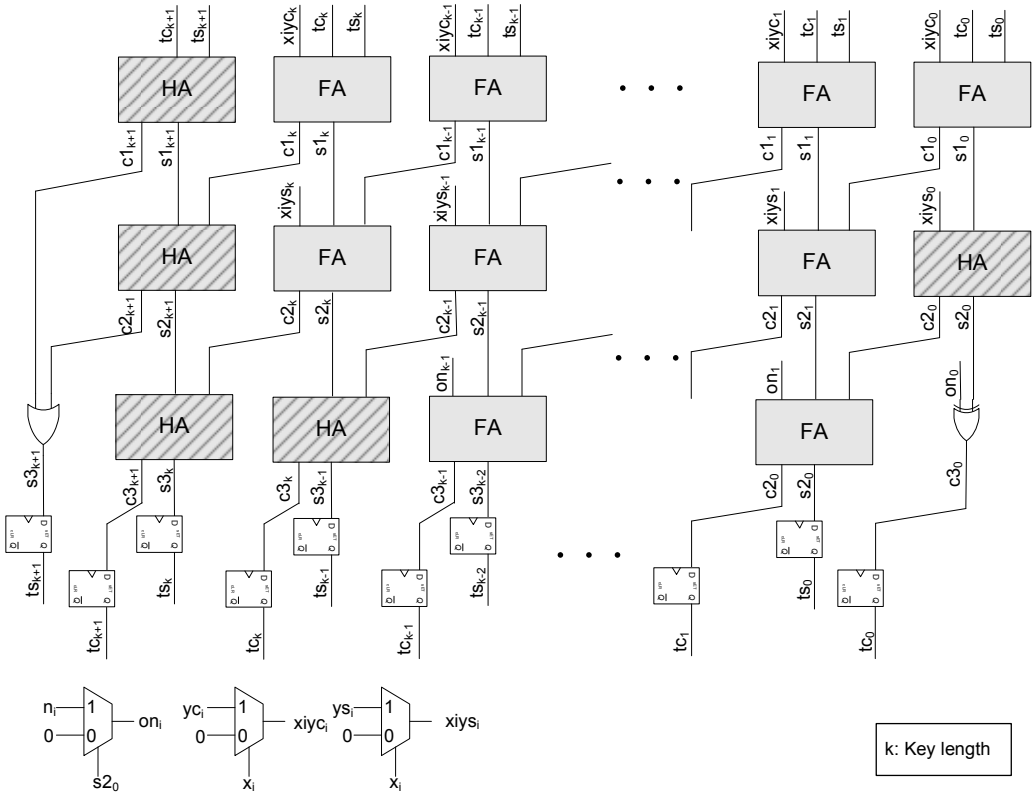

Figure 6.4: HW implementation of the Montgomery Multiplication unit using CSAs

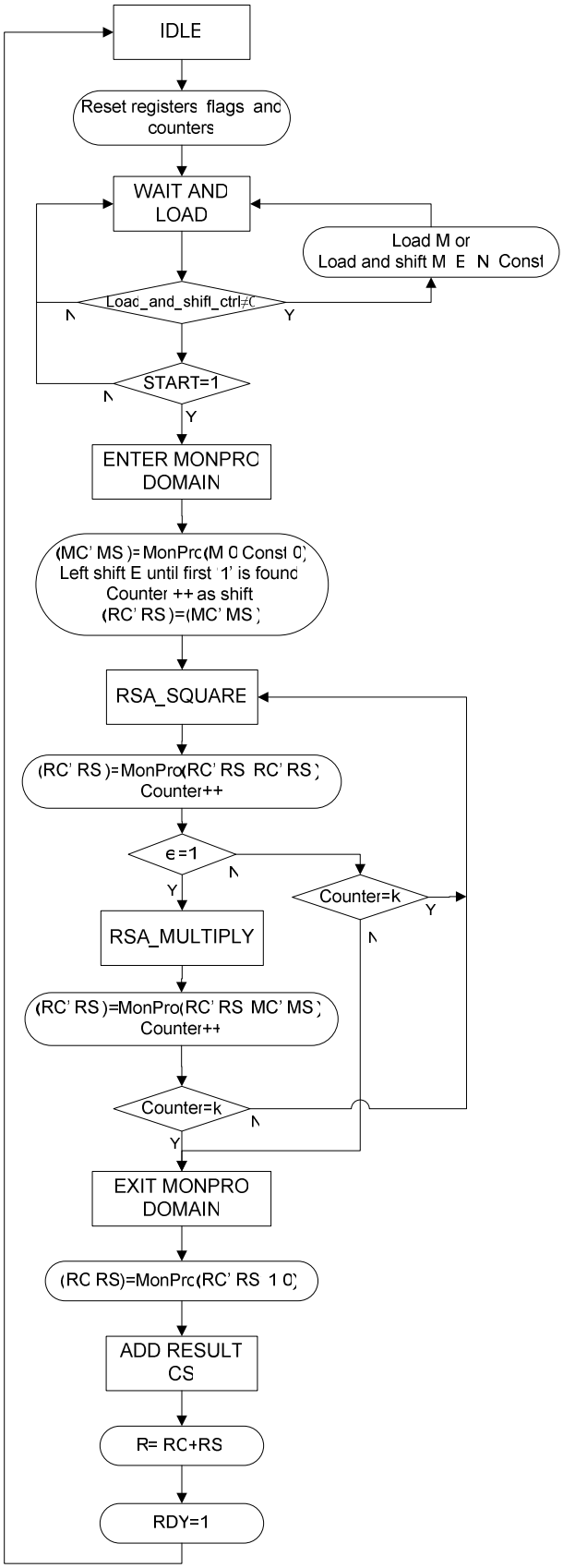

Figure 6.5: State machine of RSA main block

MonExp_NFS_CSA is implemented with a finite state machine as given in Figure 6.5. This algorithm has four inputs as M, E, N, and the constant number. E, N, and Const= $2^{2k+4} \bmod N$ do not change for every encryption, only M does. Hence there are two loading options: load all inputs or load M only. Afterwards a Start signal is waited. Then the algorithm enters the Montgomery domain and calculates (M'C,M'S) from M and Const using MonPro_NFS_CSA. It scans until the leftmost nonzero bit of the exponent and continues with squaring. The square and multiply process is continued until all the bits of E are scanned. Then the MonPro domain is to be exited by doing Montgomery multiplication on the current result (R'C,R'S) and 1. The result is still a carry save pair (RC,RS) afterwards. RC and RS are added using the CRPA. The exponentiation result is ready when this final addition is over.

6.1.2 Software for Verification

The software model which was used for verification has been realized exactly to match the steps implemented in hardware. The software code, like the hardware code, has been written using generic sizes. This has given the chance to test the implementation with 32 bit key size on the first hand. The large operand sizes have been realized with arrays of 32 bit element size. The software supports the multiples of 32 as the key size: 32, 64, 128, 256, 512, 1024, etc.

The verification of the software model itself has been checked by decrypting the encrypted data and comparing the plaintext with the decrypted text. The software takes plaintext input files, encrypts them, verifies them with decryption and creates a ciphertext output file. The steps can be seen in Figure 6.6. These files are to be used for simulation and measurements.

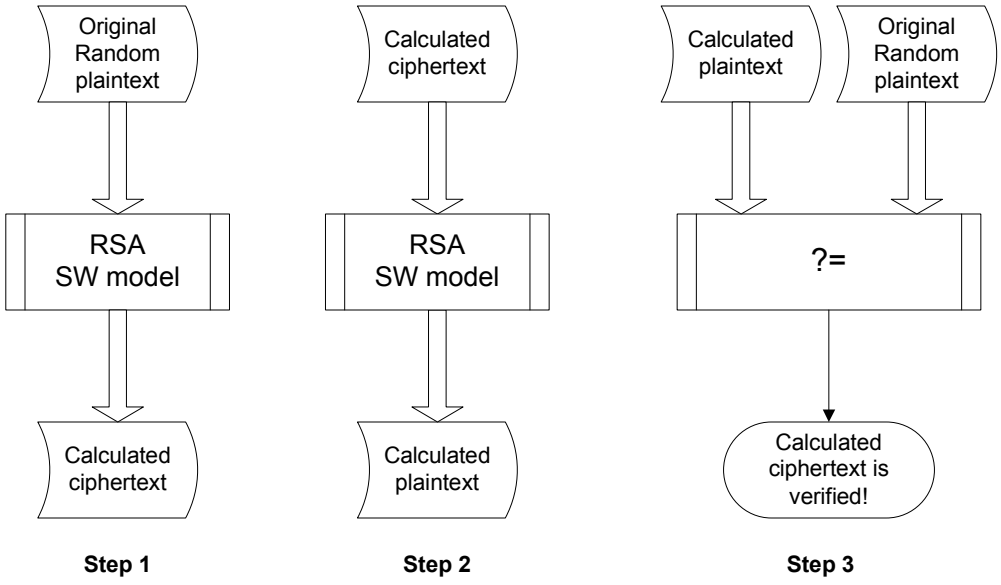

Figure 6.6: Software verification of input and output pairs

The software model is written with C/C++ language and it was compiled using Visual Studio .NET 2003.

6.1.3 Measurement

This unprotected implementation is expected be resistant to revealing the secret key in an SPA attack, because there are no different modules for squaring and multiplication. However this implementation is expected to be unprotected against SPA attacks that reveal the $H(E)$ and DPA attacks that reveal the secret key. Figure 6.7 summarizes the steps of the measurement flow.

One plaintext and one measurement is enough for an SPA attack. For implementing an SPA attack in an RSA cryptosystem, the square and multiply power consumption patterns have to be distinguished. This is done by looking at a single measurement output.

On the other hand, tens of thousands of random plaintext inputs are given to an RSA cryptosystem to make a DPA attack and retrieve the private-key.

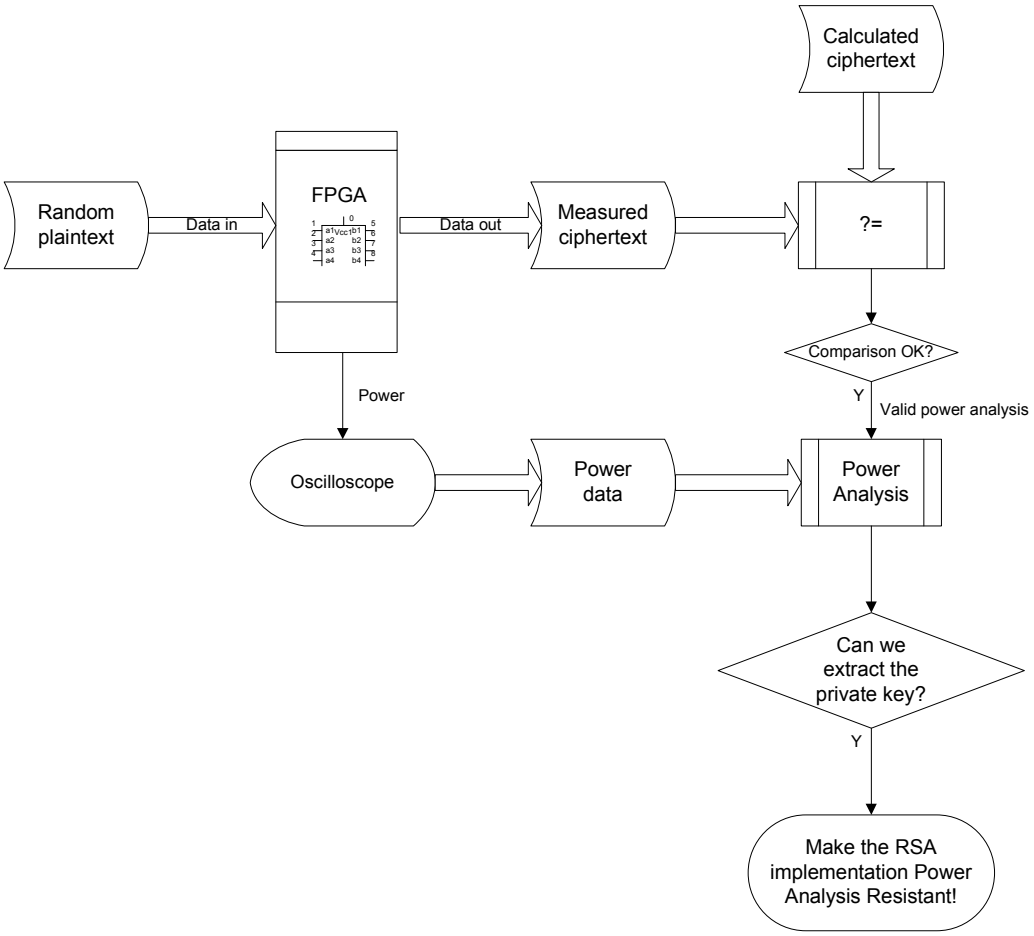

Figure 6.7: Measurement of DPA resistancy

We have tried to perform the measurements described here. Since the measurement setup was not ready, this step took more time than expected. In order to complete the rest of the thesis work, this step has eventually been skipped.

6.1.4 Implementation Results

MonPro_NFS_CSA takes $k+2$ clock cycles. The maximum frequency of the implementation with Xilinx XC2V1500 for $k=512$ is 140,96 MHz, which takes 3,65 µs resulting in a throughput rate of 140,41 Mb/s. When implemented on Xilinx XC2V4000 for *k*=1024, the maximum frequency achieved becomes 129,05 MHz; the total time 7,95 µs, and the throughput rate 128,80 Mb/s. As shown in Table 6.1, the resulting throughput rates are faster than [40-42], and almost the same speed as [43], which are also architectures using CSAs to realize Montgomery multipliers.

Table 6.1: Montgomery Multiplier implementations in comparison to previous works

Design	**Device**	**Bit length (k)**	**Clock speed (MHz)**	**Area (Slices)**	**Throughput Rate (Mb/s)**
This work	XC2V1500	512	140,96	4339	140,41
	XC2V4000	1024	129,05	5509	128,80
[40]	XC2V1500	512	72,1	3125	71,82
[41]	XC2V1500	512	105,57	4962	105,36
[42]	XC2V1500	512	126,71	5170	126,46
[43]	FPGA	1024	129,1	3611	129

Addition with CRPA takes k/w clock cycles. The decision to choose the word length w was done according to the optimum frequency of the synthesis results (See Table 6.2). In order not to make the exponentiation slower than the Montgomery Production block, *w*=16 was chosen.

The whole RSA module, MonExp_NFS_CSA takes $\left(k^2+3k+k/w+2\right)$ clock cycles for the best case where the exponent is $E=2^{k-1}$, and $\left(2k^2+4k+k/w\right)$ clock cycles for the worst case where the exponent is $E=2^k-1$. The average for the exponentiation is $\left(\frac{3}{2}k^2+5k+k/w+4\right)$ clock cycles. Table 6.3 shows the

implementation results of the Montgomery multiplier modules, and the top level RSA modules.

Table 6.2: Synthesis results of the CRPA module on XC2V1500

Key length (bits)	**Word size (bits)**	**Time (clock cycles)**	**Area (Slices)**	**Clock Speed (MHz)**
512	32	16	976	145,73
512	16	32	932	179,87

Table 6.3: Implementation results for Montgomery and RSA (top level) modules

Design module	**Para-meters**	**Time (Clock cycles)**	**Time (Clock cycles)**	**Area (Slices)**	**Clock Speed (MHz)**	**Through-put rate (b/s)**
MonPro (XC2V1500)	k=512	k+2	514	4339	140,96	140,41 M
MonPro (XC2V4000)	k=1024	k+2	1026	5509	129,05	128,80M
RSA (XC2V2000)	k=512, w=16	$3/2k^2 + 5k + k/w + 4$ (average)	395812	10240	116,35	150,50 K
RSA (XC2V6000)	k=1024, w=16	$3/2k^2 + 5k + k/w + 4$ (average)	1578020	25193	84,33	54,72 K

For our first unprotected RSA implementation with k=512 and w=16, we get an average of 395812 clock cycles. The maximum frequency of the implementation with Xilinx XC2V2000 is 116,35 MHz, which takes an average of 3,4 ms for the whole exponentiation process, giving us a throughput rate of 150,50 Kb/s for the average case. For the best case, the exponentiation takes 263712 clock cycles resulting in 2,27 ms. The unprotected RSA implementation has been repeated for 1024 bits.

Using the parameters as *k*=1024 and *w*=16, we get an average of 1578020 clock cycles. Implemented on Xilinx XC2V6000, the maximum frequency becomes 84,33 MHz, whilst the average time for exponentiation becomes 18,71 ms resulting in a throughput rate of 54,72 Kb/s. For the best case, the exponentiation takes 1051712 clock cycles which is 12,47ms.

6.2 RSA Cryptosystem Implementation Immune to Power Analysis Attacks

For the RT-WM algorithm (Chapter 5.4.1), which is applied as a countermeasure against DPA attacks in this study, the number of items in the $\omega[i]$ array is:

$$\omega\_count = \lceil (k-b)/t \rceil \quad \textbf{(6.1)}$$

This gives us the number of ω_count comparisons and subtractions in preprocessing phase 1.

One comparison takes one clock cycle and since the existing CRPA is used in subtractions, one subtraction costs w (word count of CRPA) clock cycles.

The 2nd phase of the preprocessing calculates $M^r \bmod N$, $M^{dm} \bmod N$, and $M^{2^b} \bmod N$. It takes $(2^b - 1)$ MonPro calculations for this phase.

The 3rd phase of the preprocessing finalizes the table. The table has 2^t k-bit items and it takes $(2^t - 1)$ MonPro calculations to finish the table. Since one MonPro calculation takes $(k+2)$ clock cycles in the proposed design, the total time spent in the preprocessing calculations becomes $\lceil (k-b)/t \rceil \cdot (CRPA\_word\_count + 1) + (2^b + 2^t - 2) \cdot (k+2)$ clock cycles as shown in Table 6.4.

Table 6.4: Preprocessing time equations of RT-WM algorithm

Preprocessing	**Time (clock cycles)**
Prep. Phase 1	$\lceil (k-b)/t \rceil \cdot (w+1)$
Prep. Phase 2	$(2^b - 1) \cdot (k+2)$
Prep. Phase 3	$(2^t - 1) \cdot (k+2)$
Total	$\lceil (k-b)/t \rceil \cdot (w+1) + (2^b + 2^t - 2) \cdot (k+2)$

The RT-WM parameters selected for this study and the resulting additional time are shown in Table 6.5. The exponentiation method which replaces the square and multiply method now becomes like *t* times square and multiply once with a table

value. A final multiplication is needed for the normalization. Therefore, accepting that $\omega_0 \neq 0$ for *k*-bit exponents, the exponentiation time achieved is

$(\omega\_count-1)\cdot(t+1)+1=\lceil(k-b)/t\rceil\cdot(t+1)+1$ Montgomery multiplications.

Table 6.5: Preprocessing time of RT-WM for the implementation values

RT-WM algorithm	**Key length (bits)**	**b (bits)**	**t (bits)**	**CRPA word count**	**Time (clock cycles)**
Pre-processing	512	3	2	16	9492

Since the zero windows are not skipped here, which is different than the *m*-ary method, the best case, the average case and the worst case exponentiation time in RT-WM method are the same. In addition to the mentioned preprocessing, 2 multiplications are needed for entering and exiting the MonPro domain (Algorithm 5.1) and k/w clock cycles are needed for CRPA addition. Table 6.6 shows the exponentiation time and the total time spent in RT-WM algorithm.

The total time required by the new algorithm, realized with 512-bit key length, 2-bit window length, and a 3-bit random number, needs 404276 clock cycles and brings an overhead of 11,8% in total time (in clock cycles), when compared to the *m*-ary method. The *m*-ary method needs an average of 703,25 multiplications (See Table 4.2), which makes 361471 clock cycles. The reason why we compare this result with the results of the *m*-ary exponentiation method, is that both methods use *t* size windows, where $m=2^t$. This preprocessing brings an overhead of 2,1% in total time when compared to the binary method.

Table 6.6: RT-WM exponentiation and total time

Exp. Time (clock cycles) (parametric)	**Exp. Time (clk cycles) (k=512, b=3, t=2)**	**Total Time (clk cycles)**
$(\lceil(k-b)/t\rceil\cdot(t+1)+3)\cdot(k+2)+k/w$	394784	404276

6.2.1 Hardware Implementation

Figure 6.8 shows the state machine of the RT-WM implementation of the RSA cryptosystem. As it can be seen, new states have been added: *Preprocess 1*, *Preprocess 2, Preprocess 3,* and *Normalize* – which are shown on the right side of the figure. The time spent, in these additional states has been explained in Chapter 6.2. Preprocessing Phase 1, where Ew is calculated, is done before entering the MonPro domain. Preprocessing Phases 2 and 3 are done in order to fill in the randomized table. RSA_Multiply and RSA_Square states have changed with respect to the former implementation.

Now RSA_Square does *t* times squaring consecutively, once the state is entered. RSA_Multiply is not done with *M;* the corresponding table entry is used instead. There is a final multiplication state after the $\omega[i]$ array is scanned. This multiplication applies to the normalization step. Afterwards the state machine enters the Exit_MonPro state, and the rest is followed as stated in the former state machine, shown in Figure 6.5.

6.2.2 Implementation Results

The implementation results of the RT-WM algorithm, realized with 512-bit key length, 2-bit window length, and, a 3-bit random number, on Xilinx XCV2600E, are shown in Table 6.7. An exponentiation time of 18,43 Kb/s throughput and an area of 22712 slices are achieved. The maximum clock frequency is 14,55 MHz. The total encryption process takes 27,79 ms, which was 3,4 ms for the unprotected implementation.

Table 6.7: Implementation results for RSA with RT-WM

Design module	**Parameters**	**Time (Clock cycles)**	**Area (Slices)**	**Clock Speed (MHz)**	**Throughput rate (Kb/s)**
RSA (XCV2600E)	*k*=512, *w*=16, b=3, t=2	404276	22712	14,55	18,43

The unprotected implementation fits into XCV1000E, occupying 9037 slices, which is 73% of the available slices. When implementing the protected architecture, a major modification is done in the state machine (Figure 6.8); but the main hardware need is 6 pair of *k*-bit registers due to the RT-WM algorithm (Algorithm 5.1).

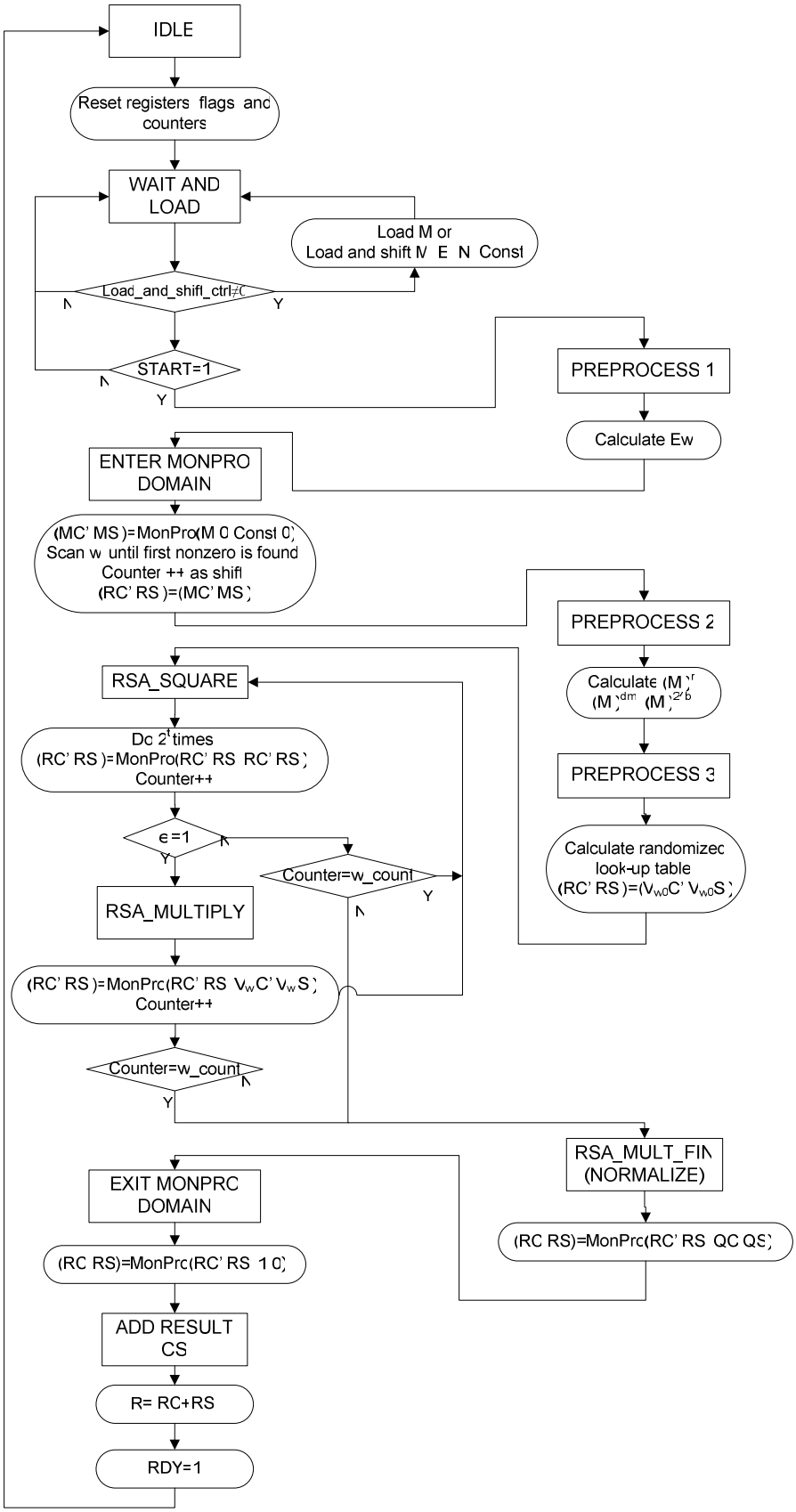

Figure 6.8: State Machine of RT-WM implementation of RSA

As there are two registers in each slice of Virtex-E family, this need causes an inefficient use of the slices which prevents fitting into the same device. The number of slices are 2,5 times the unprotected implementation. Thus the routing also becomes inefficient causing a great decrease in the speed (Table 6.7).

6.3 Optimization of Hardware Implementation

The measurement setup includes the FPGA XCV1000E, from the Xilinx Virtex-E family. The previously mentioned implementation results of the unprotected design were realized on FPGA devices from the Xilinx Virtex-II family, to be able to compare with the previous designs in the literature, which were also implemented on Xilinx Virtex-II family.

Table 6.8: All implementation results on Virtex-E family devices

Design	Unprotected RSA	Protected RSA	Protected RSA
Device	XCV1000E	XCV2600E	XCV1000E
Parameters	k=512, w=16	k=512, w=16, b=3, t=2	k=512, w=16, b=3, t=2
Block RAM (CountxEntryxWidth)	No	No	2x4x513
Area (slices)	9037	22712	10986
Time (clock cycles)	395812	404276	404276
Clock Speed (MHz)	81,06	14,55	66,66
Throughput rate (Kbit/s)	104,85	18,43	84,42
Exponentiation time (ms)	4,88	27,79	6,06

In order to ensure future measurements of the unprotected and protected designs accomplished throughout this study, these designs were implemented on Xilinx Virtex 1000E, too. The unprotected design fit into the XCV1000E occupying 9037 slices, which is 73% of the available slices. Meanwhile, the protected design needed

22712 slices which could fit into the XCV2600E. Therefore the protected design needed an optimization to become measurable with the available measurement setup.

Virtex-E family FPGAs incorporate large block SelectRAM memories, where the data widths of the ports can be configured, and the routing is optimized. Hence we used these built-in block RAM structures for the protected design in order to fit into the XCV1000E. The RT-WM algorithm needs 8x513 bits to be used as the “randomized table” values for the chosen parameters (Chapter 6.2), which were realized with registers. One needs to separate the carry and save pairs in different RAM blocks in order to have read/write access to them at the same clock cycle. Therefore two RAM blocks of 513-bit data length and 4 entries have been defined.

The resulting implementation fit into the device occupying 10986 slices, as 89% of the available slices. All implementation results on Virtex-E family devices are given in Table 6.8. Comparing the protected RSA implementations, we see that the clock speed increased from 14,55 MHz to 66,66 MHz, making the average case throughput increase from 18,48 Kb/s to 84,42 Kb/s. Total exponentiation time is reduced from 27,11 ms to 6,06 ms. The time and area cost of the protected design is reduced with block SelectRAM usage.

7. RESULTS AND FUTURE WORK

We have implemented RSA cryptosystem by using Montgomery multiplier and all the additions in the Montgomery multiplier are performed by Carry Save Addition (CSA). CSA is an appropriate way of reducing 3-*k* bit operands to 2-k bit operands. Hence, throughout the algorithm, each number is represented by a pair as sum and carry. At the end of the square and multiply algorithm the numbers in the resulting pair are added to form the result. We give the comparisons with the previous Montgomery multiplier architectures, which also used CSAs. Our implementation is faster than the compared architectures except one, which is almost the same speed as ours [39].

The second architecture of this study has made the cryptosystem resistant against DPA attacks. With the final optimization using block SelectRAM structures, the total time has increased by 24,2% with respect to the unprotected implementation, while the throughput rate decreased by 19,5%. Thus, the final protected implementation became DPA resistant, still fitting into the same device, but slower.

The aim of the optimization was in fact, to enable the future work mentioned below.

Following the implementation results described in this thesis, a number of projects could be taken up to accomplish the following:

- The measurement setup completion of the unprotected implementation
- Implementing an SPA attack on the unprotected implementation to prove that the Hamming weight of the exponent can be extracted
- Applying “Always Square & Multiply Method” upon the unprotected implementation against SPA attacks and implementing a DPA attack on the implementation in the previous item to prove that the secret key can be extracted
- Implementing a DPA attack against the protected implementation to prove that the secret key cannot be extracted
- The design will be improved according to the attack results.

REFERENCES

[1] **Rivest, R.L., Shamir, A., and Adleman, L.**, 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communications of the ACM*, **21**, 120-126.

[2] **Koç, Ç.K.**, 1994. High-Speed RSA Implementation, *RSA Laboratories Technical Report*, Redwood City, California, USA.

[3] **Montgomery, P.L.**, 1985. Modular Multiplication without Trial Division, *Mathematics of Computation*, **44**, 519-521.

[4] **Tinder, R.F.**, 2000. Engineering Digital Design: Revised Second Edition Academic Press, San Diego.

[5] **Örs, S.B.**, 2005. Hardware Design of Elliptic Curve Cryptosystems and Side-Channel Attacks, *PhD Thesis*, Katholieke Universiteit Leuven, Leuven.

[6] **Kocher, P., Jaffe, J., and Jun, B.**, 1999. Differential Power Analysis, *Proceedings of Advances in Cryptography-CRYPTO'99*, Lecture Notes in Computer Science, Santa Barbara, USA, **1666**, pp. 388-397, Springer-Verlag.

[7] **Itoh, K., Yajima, J., Takenaka, M., and Torii, N.**, 2003. DPA Countermeasures by Improving the Window Method, *Cryptographic Hardware and Embedded Systems (CHES)*, Lecture Notes in Computer Science, California, USA, August 2003, **2523**, pp. 303-317, Springer-Verlag.

[8] **Pawlan, M.**, 1998. Cryptography: The Ancient Art of Secret Messages, *Sun Developer Network*, http://java.sun.com/developer/technicalArticles/ Security/Crypto.

[9] **Menezes, A.J., Van Oorschot, P.C., and Vanstone, S.A.**, 1996. Handbook of Applied Cryptography, CRC Press.

[10] **Stinson, D.R.**, 2002. Cryptography Theory and Practice, Chapman & Hall/CRC, Waterloo, Ontario.

[11] **Diffie, W. and Hellman, M.E.**, 1976. New Directions in Cryptography, *IEEE Transactions on Information Theory*, **22**, 644-654.

[12] **Knuth, D.E.**, 1981. The Art of Computer Programming: Seminumerical Algorithms, Addison-Wesley, Reading.

[13] **Bos, J. and Coster, M.**, 1989. Addition Chain Heuristics, *Advances in Cryptology - CRYPTO 89*, Lecture Notes in Computer Science, Santa Barbara, California, USA, **435**, pp. 400-407, Ed. Brassard, G., Springer-Verlag.

[14] **Koç, Ç.K.**, 1995. Analysis of Sliding Window Techniques for Exponentiation, *Computers and Mathematics with Applications (CANDM)*, **30**, 17-24.

[15] **Batina, L., Örs, S.B., Preneel, B., and Vandewalle, J.**, 2003. Hardware Architectures for Public Key Cryptography, *The VLSI Journal Integration*, **34**, 1-64, Elsevier Science Publishers B. V.

[16] **Walter, C.D.**, 1999. Montgomery Exponentiation Needs No Final Subtraction, *Electronic Letters*, **35**, 1831-1832.

[17] **Dhem, J.-F., Koeune, F., Leroux, P.-A., Mestre, P., Quisquater, J.-J., and Willems, J.-L.**, 1998. A Practical Implementation of the Timing Attack, *Universit'e Catholique de Louvain Crypto Group Technical Report*, Belgium.

[18] **Brown, S.D. and Vranesic, Z.G.**, 2003. Fundamentals of Digital Logic with Verilog Design, McGraw-Hill, Toronto.

[19] **Kömmerling, O. and Kuhn, M.G.**, 1999. Design Principles for Tamper Resistant Smartcard Processors, *Proceedings of the USENIX Workshop on Smartcard Technology*, Chicago, Illinois, USA, May 1999, pp. 9-20.

[20] **Boneh, D., DeMillo, R.A., and Lipton, R.J.**, 1997. On the importance of checking cryptographic protocols for faults, *International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT '97)*, Lecture Notes on Computer Science, Konstanz, Germany, **1233**, pp. 37-51, Springer-Verlag.

[21] **Joye, M., Lenstra, A.K., and Quisquater, J.-J.**, 1999. Chinese remaindering based cryptosystem in the presence of faults, *Journal of Cryptology: The journal of the International Association for Cryptologic Research*, **12**, 241-245.

[22] **Kocher, P.**, 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, *Advances in Cryptology - CRYPTO '96*, Lecture Notes on Computer Science, **1109**, pp. 104-113, Springer-Verlag.

[23] **Peeters, E., Standaert, F.-X., and Quisquater, J.-J.**, 2007. Power and Electromagnetic Analysis: Improved Model, Consequences and Comparisons, in *The VLSI Journal Integration*, Special Issue of Embedded Cryptographic Hardware, **40**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands.

[24] **Shamir, A. and Tromer, E.**, 2004. Acoustic Cryptanalysis on Nosy people and Noisy Systems: Preliminary proof-of-concept presentation, http://www.wisdom.weizmann.ac.il/~tromer/acoustic.

[25] **Messerges, T.S.**, 2000. Power analysis attack countermeasures and their weaknesses, *Communications, Electromagnetics, Propagation and Signal Processing Workshop*, Illinois, USA, October 2000.

[26] **Coron, J.S.**, 1999. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems, *Cryptographic Hardware and Embedded Systems: First International Workshop, CHES'99*, Lecture Notes in Computer Science, Worcester, MA, USA, August 1999, **1717**, pp. 292-302, Eds. Koç, Ç.K. and Paar, C., Springer-Verlag.

[27] **Shamir, A.**, 2000. Protecting smart cards from passive power analysis with detached power supplies, *Cryptographic Hardware and Embedded Systems (CHES)*, Lecture Notes in Computer Science, Worcester, Massachusetts, USA, **1965**, pp. 71-77, Eds. Koç, Ç.K. and Paar, C., Springer-Verlag.

[28] **Walter, C.D.**, 2002. MIST: An Efficient, Randomized Exponentiation Algorithm, *Topics in Cryptology - CT-RSA 2002: The Cryptographer's Track at the RSA Conference 2002*, Lecture Notes in Computer Science, San

Jose, CA, USA, February 2002, **2271**, pp. 53-66, Ed. Preneel, B., Springer-Verlag.

[29] **Okeya, K. and Takagi, T.**, 2003. The Width-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure Against Side Channel Attacks, *Topics in Cryptology - CT-RSA 2003: The Cryptographers' Track at the RSA Conference*, Lecture Notes on Computer Science, San Francisco, CA, USA, April 2003, **2612**, pp. 328-342, Springer-Verlag.

[30] **Chevallier-Mames, B.**, 2004. Self-Randomized Exponentiation Algorithms, *Topics in Cryptology – CT-RSA 2004*, **2964**, 236-249, Lecture Notes in Computer Science, Springer-Verlag.

[31] **Goubin, L. and Patari, J.**, 1999. DES and differential power analysis the "duplication" method, *Cryptographic Hardware and Embedded Systems (CHES)*, Lecture Notes in Computer Science, Worcester, Massachusetts, USA, **1717**, pp. 158-172, Eds. Koç, Ç.K. and Paar, C., Springer-Verlag.

[32] **Messerges, T.S.**, 2002. Securing the AES finalists against power analysis attacks, *Proceedings of the 7th International Workshop on Fast Software Encryption (FSE)*, Lecture Notes in Computer Science, New York, NY, USA, April 2000, **1978**, pp. 150-164, Ed. Schneier, B., Springer-Verlag.

[33] **Clavier, C. and Joye, M.**, 2001. Universal Exponentiation Algorithm, *Cryptographic Hardware and Embedded Systems CHES 2001: Third International Workshop*, Lecture Notes in Computer Science, Paris, France, May 2001, **2162**, pp. 300-308, Eds. Koç, Ç.K., Naccache, D., and Paar, C., Springer-Verlag.

[34] **Solinas, J.A.**, 2000. Efficient Arithmetic on Koblitz Curves, in *Design, Codes and Cryptography*, Special issue on Towards a quarter-century of public key cryptography, **19**, pp. 195-249, Kluwer Academic Publishers, Norwell, MA, USA.

[35] **Möller, B.**, 2001. Securing Elliptic Curve Point Multiplication against Side-Channel Attacks, *Proceedings of the 4th International Conference on Information Security*, Lecture Notes on Computer Science, **2200**, pp. 324-334, Eds. Davida, G.I. and Frankel, Y., Springer-Verlag.

[36] **Miller, V.S.**, 1985. Use of Elliptic Curves in Cryptography, *Advances in Cryptology - CRYPTO 85*, Lecture Notes in Computer Science, Santa Barbara, California, USA, August 1985, **218**, pp. 417-426, Ed. Williams, H.C., Springer-Verlag.

[37] **Yen, S.-M., Chen, C.-N., Moon, S., and Ha, J.**, 2004. Improvement on Ha-Moon Randomized Exponentiation Algorithm, *Information Security and Cryptology (ICISC 2004)*, Lecture Notes on Computer Science, Seoul, Korea, **3506**, pp. 154-167, Springer-Verlag.

[38] **Izu, T., Möller, B., and Tsuyoshi, T.**, 2002. Improved Elliptic Curve Multiplication Methods Resistant against Side Channel Attacks, *Proceedings of Indocrypt 2002*, Lecture Notes on Computer Science, Hyderabad, India, December 2002, **2551**, pp. 296-313, Springer-Verlag.

[39] **Alptekin Bayam, K., Örs, S.B., and Örencik, B.**, 2007. A Hardware Implementation of RSA, *International Conference on Security of*

Information and Networks - SIN2007, Gazimagusa, North Cyprus, May 2007.

[40] **Manochehri, K. and Pourmozafari, S.**, 2005. Modified Radix-2 Montgomery Modular Multiplication to Make It Faster and Simpler, *International Conference on Information Technology: Coding and Computing*, Las Vegas, Nevada, USA, **1**, pp. 598 - 602, IEEE.

[41] **McIvor, C., McLoone, M., and McCanny, J.V.**, 2003. Fast Montgomery modular multiplication and RSA cryptographic processor architectures, *37th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, USA, November 2003, **1**, pp. 379-384, IEEE.

[42] **McIvor, C., McLoone, M., and McCanny, J.V.**, 2004. Modified Montgomery modular multiplication and RSA exponentiation techniques, *Proceedings of Computers and Digital Techniques*, **151**, 402-408.

[43] **Fournaris, A.P. and Koufopavlou, O.**, 2005. A new RSA encryption architecture and hardware implementation based on optimized Montgomery multiplication, *International Symposium on Circuits and Systems (ISCAS 2005)*, Kobe, Japan, May 2005, **5**, pp. 4645-4648, IEEE.

BIOGRAPHY

Keklik Alptekin Bayam was born in Giresun, Turkey in 1980. She graduated from Bursa Anatolian High School in 1998. In 2002, she received B.Sc. degree in Electronics and Communication Engineering from Istanbul Technical University. In 2003, she started the Computer Engineering M.Sc. program in Istanbul Technical University. She is currently working as a digital design engineer in STMicroelectronics. Her research interests are digital design, cryptography, and programming.