Incorporating Error Detection in an RSA Architecture

L. Breveglieri[1], I. Koren[2], P. Maistri[1], and M. Ravasio[3]

[1] Department of Electronics and Information Technology, Politecnico di Milano, Milano, Italy
{brevegli, maistri}@elet.polimi.it
[2] Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, USA
koren@ecs.umass.edu
[3] STMicroelectronics, Agrate Brianza, Milano, Italy
moris.ravasio@st.com

Abstract. Most successful attacks against hardware implementations of cryptographic systems make use of side-channel information leakage. Recently, some attacks have been proposed against various cryptosystems, which exploit deliberate error injection during the computation process. Several error detection schemes have been proposed in order to counteract these attacks. In this paper, we add a residue-based error detection scheme to an RSA architecture and evaluate the area and latency overheads with respect to the basic architecture.

1 Introduction

Hardware implementations of cryptographic systems have become very popular, in order to satisfy the latest demands in terms of performance and tamper resistance. The most widely adopted public-key algorithm is currently the RSA cryptosystem (proposed by Rivest, Shamir and Adleman) that relies on the difficulty in factorizing large integers.

In the past, most attacks were aimed at solving the factorization problem. However, RSA uses currently 1024-bit operands and factorization of such large integers is unaffordable with current computational power. An alternative way to attack a cipher is through attempts to break a specific implementation by finding a correlation between physical information leakage and the secret keys (e.g., simple and differential power analysis, timing attacks). Recently, new side channel attacks have been proposed. In [4], the authors showed how deliberate hardware faults can be exploited to break a cryptographic algorithm and retrieve the key. They have addressed public-key schemes in general and provided examples, including a description of an attack against RSA. Attacks against RSA were later refined in [1] where the authors showed how a single faulty ciphertext can be used to easily factor the RSA modulus, thus breaking the cryptosystem. It should be pointed out that RSA implementations based on the Chinese Remainder Theorem (CRT) can be broken more easily than a basic implementation [2,4]. Other attacks against CRT-RSA appear in [13] and [14].

L. Breveglieri et al. (Eds.): FDTC 2006, LNCS 4236, pp. 71–79, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Fault-based attacks are highly effective, since few carefully localized errors can break the cipher. While attacking regular RSA requires the ability to inject single-bit errors, it must be noted that the only requirement to successfully break CRT-RSA is that *only one* of the two sub-signatures is corrupted. Hence, if the attacker can inject any error into one sub-exponentiation and have a result back, he can break the cryptosystem. This implies that the error model could be as general as possible.

Although they may have been considered to be of only theoretical value, some initial experiments have shown that such attacks are possible in practice. Therefore, several countermeasures were proposed to foil the attacks. In 1999, Shamir registered a patent [10] where a multiplicative masking is used against timing and fault attacks. In 2000, Walter [12] has suggested to use residue codes to protect modular arithmetic operations. The residue code can help detect both transient and permanent faults. The error coverage depends on the value of the base modulus that is chosen: higher values of the base modulus allow higher detection rates. The overhead of the residue code is approximately the cost of an extra digit in the operands. In general, this overhead can be in terms of area if an extra element is included in each functional unit or in terms of time when the same functional unit is reused. In 2004, Gueron [5] described an extended version of the modular exponentiation and Montgomery multiplication algorithms by using a residue code in multi-digit algorithms. If the base modulus is chosen to be the maximum value of a digit (i.e., $2^d - 1$, where d is the digit size in bits), generation is very simple [8,12].

In this paper, we present a reasonably simple architecture for computing the RSA with protection against injected faults using residue codes. Residue codes were chosen since they can protect arithmetic operations quite efficiently. Moreover, if a specific fault pattern is likely to occur, a sufficiently large base modulus can be chosen in order to provide good coverage. We adapt the suggestion made by Walter [12] to a specific architecture and evaluate the benefits and overheads. We also show how to further improve this specific solution.

In Section 2 we briefly describe the RSA cryptosystem and our reference architecture. In Section 3 we present the detection mechanism by means of residue codes. Section 4 provides the details of our implementation of a circuit with error detection capabilities, detailing the implementation choices. Finally, Section 5 concludes the paper.

2 The RSA Cryptosystem

The RSA cryptosystem [9] is based on few essential parameters, namely, the public moduli $N = p \cdot q$, where p and q are two large primes, each $n/2$ bits long; d, the private exponent key; and e, the public exponent key, selected such that $e \cdot d = 1 \mod (p-1)(q-1)$. Encryption of a message m is done by computing $m^e \bmod N$. Decryption is computed by another exponentiation, namely $(m^e)^d \bmod N$. The most critical operation is therefore the modular exponentiation. A large number of multiplications is needed to perform exponentiation and

quite often, a simple Square-and-Multiply algorithm is used. Various exponentiation algorithms are described in [7]. Most implementations of exponentiation use the modified Montgomery multiplication algorithm (depicted in Figure 1), since it avoids the use of expensive trial divisions and avoids conditional branches which might benefit side channel attacks at the cost of few additional iterations, while maintaining the correctness of the algorithm [11]. Many different architec-

Input:	$X = (0\,0\,X_n \cdots X_1 X_0)$
	$Y = (Y_n \cdots Y_1 Y_0)$
	$M = (M_{n-1} \cdots M_1 M_0)$
Output:	$X \cdot Y \cdot 2^{-(n+2)} \bmod M$
1.	$U \leftarrow 0$
2.	for i from 0 to $(n+2)$ do
a	if $(U_0 = 1)$ then $U = U + M$
b	$U = (U/2) + X_i Y$
3.	return U

Fig. 1. The modified Montgomery Multiplication, radix-2 [6]

tures have been proposed, differing mostly in some minor modifications to the Montgomery algorithm and in the digit size. The designer can thus obtain the desired trade-off between area and latency.

Our selected architecture was inspired by the design presented in [6]. Our solution differs from the original design mainly in the absence of the Z-processor, i.e., the squaring functional unit; this modification has also been suggested by the authors of [6]. The basic implementation includes a core (consisting of a *control unit* and a *Processing Element* (PE)), an internal memory and a bridge. The architecture is highly parameterized and it can therefore support future operand sizes. A small area is achieved by using a serial-digit approach. The exponentiation is computed by using the Square-and-Multiply algorithm; each multiplication scans the multiplier one bit at a time, and computes the result by using the Double-and-Add algorithm. Finally, each addition is performed by computing the result one digit at a time.

The PE is able to perform the basic required operations (addition, subtraction and addition with extra shifting) on the n-bit operands by repeated steps. The word size of the computational core is only 25% of the memory word size, which is in turn considerably smaller than the size of the modulus N. This allows to reduce the PE critical path and achieve higher frequencies. Accessing the memory implies a certain latency, due to the signal setup time and the register layers required to obtain stable values at the PE inputs. However, since the PE word size is 25% of the memory word size, producing a single result requires 4 clock cycles to fetch the next memory read. This is done at each iteration except the last one, when the PE is processing the last word.

3 Online Detection

In this section, we describe our detection scheme for errors injected during an RSA operation. The underlying principle is associating check bits to the data we are processing. If we are able to maintain the relationship between the data and the corresponding check bits throughout the entire process, the final result and the associated check bits should still satisfy the same relationship. This can be accomplished by propagating the check bits according to a specified set of prediction rules. When the data is processed by a specific functional unit, the corresponding check bits are processed in parallel using the associated prediction rule in order to preserve the relationship. As an example, consider a parity bit associated with a data word. When computing the exclusive OR of two different words, we can predict the parity bit of the result by XOR'ing the parity bits of the operands.

The code chosen for error detection must be simple. First of all, its generation and propagation overheads should be negligible relative to the computation of the main algorithm. Moreover, the need for prediction does not allow the use of complex codes, which may be very efficient in detecting faults but will be very expensive when implementing the various prediction circuits. The overhead of the error detection code should always be compared to brute force duplication which is the simplest way to detect errors. If detecting errors using codes is cheaper than duplication, then it is a viable solution.

To make the prediction rules as simple as possible, the code must be compatible with the operations performed on the data. Since RSA is based on modular integer arithmetic, residue codes are a natural choice. From the theory on modular arithmetic we know that the residue of the sum is the sum of the residues, possibly reduced once more:

$$(X + Y) \bmod R = ((X \bmod R) + (Y \bmod R)) \bmod R \tag{1}$$

where X and Y are two operands and R is the base modulus. For instance, take $X = 8$, $Y = 13$ and $R = 3$: $8 \bmod 3 = 2$, $13 \bmod 3 = 1$, $8 + 13 = 21$ and $21 \bmod 3 = 0$, and finally $(2) + 1 \bmod 3 = 3 \bmod 3 = 0$. A similar rule holds for subtraction and multiplication. However, since all the high-level operations are implemented in terms of simple additions, this is the only prediction rule employed in our system.

The main issue is that we have to deal with two different moduli at the same time: the modulus of the residue code, usually smaller than the word size, and the modulus used by the RSA, which in contrast, is very large. When performing a reduction of the result, the check bits have to also be modified accordingly. Extending the algorithm with residue codes is straightforward, thanks to the properties of modular arithmetic, but there are several issues that must be considered: the residue code may require an additional reduction as shown in Equation (1), and the required division by 2. The former issue is addressed by correcting the residue if it overflows the boundaries of its domain. No information is lost, since the residue is stored in a larger data register. Regarding the

latter issue, the right shift is computed in the Montgomery multiplication only after we are sure that the operand is even, by adding the modulus only to odd inputs (see algorithm in Figure 1, instruction 2.a). However, nothing can be said about the value of the residue. The residue of an even (odd) value can be either odd or even; therefore, when an odd residue must be right shifted, the (odd) residue base must be added to provide a congruent even residue value before dividing by 2.

4 Implementation

In this section we present our implementation of an RSA architecture with error detection capabilities, discuss the differences from the basic architecture and estimate the resulting overheads in terms of area and latency.

When incorporating an error detection code in the architecture, there are three new components to be considered:

1. A code generator: the check bits must be first generated from the initial raw data, possibly using a dedicated unit;
2. A set of prediction rules, needed to propagate the check bits through any operation performed in the encryption process;
3. A code validator: at the end of the encryption, the check bits must be verified against the computed data.

The code generation is obtained by using a dedicated functional unit, which is situated between the input interface and the memory. While operands are loaded into memory one word at a time, the residue generator computes and accumulates the residue into an internal register. When an additional word is loaded into memory, the partial residue value is updated. Upon deactivation of the memory write signal, the final residue value is written into the next memory word.

The check bits generated from a single word depend on the base modulus: choosing a modulus of the form $2^h - 1$ allows to compute the check bits by splitting each word into h-bit-long nibbles and adding them together. In our implementation, the size of the residue base modulus is chosen as 1/8 of the memory word size, i.e., half the PE word size. A tree of Carry-Save Adders (CSAs) is used to reduce the 9 input nibbles (8 coming from the data word, and the current value stored in the internal register) down to a single pair (Carry, Sum). However, any other size can be chosen: smaller values will give deeper trees, while larger values will require fewer steps. The size of the processing element should be considered as the upper bound, since the residue base must fit within the adder. Our choice was the largest divisor that fits the PE size, allowing a simpler design.

Although the carry output of each CSA is shifted with respect to the sum output by one bit position, the values are implicitly reduced by routing the most significant bit into the least significant position. For instance, if the size of the residue code is r, the sum output is $(s_{r-1} \dots s_0)$, while the carry output is

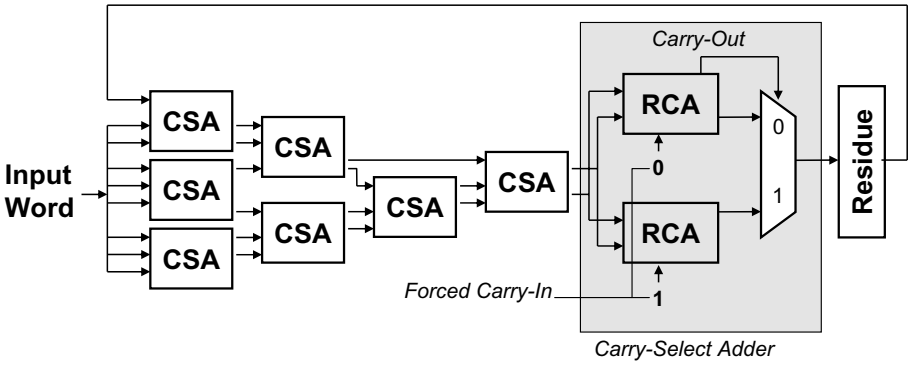

Fig. 2. Residue generation unit, with residue size being 1/8 of memory word size

$(c_r c_{r-1} \ldots c_1 0)$. The residue of the carry output can be computed in the same way (one more time) by splitting the carry into a least significant word and a single most significant bit and adding them together. Since the least significant bit of the carry vector is null, adding the most significant bit is just a simple rerouting.

The final residue generation, i.e., the summation of the carry and sum vectors is computed by using a Carry-Select Adder. In principle, the final addition might result in an $r+1$-bit-long vector. Hence, two additions are computed in parallel, forcing the carry-in to 0 or 1, respectively. Finally, the carry out of the adder with null carry-in is used to select the proper result to be stored in the internal register. Figure 2 illustrates the overall architecture of the residue generation unit. The CSA layer allows to shrink the sum of 9 operands down to only 2 addends with a delay of only a few gates. On the other hand, the twin ripple-carry adders allow to obtain a value in the residue domain within the PE delay.

The code prediction is performed within each single operation, since the rule often matches the operation itself: the residue of an addition, for instance, is the sum of the input residues. Therefore, the PE can also be used for residue calculation after the main operation is completed. This is a straightforward application of Equation (1). This simple rule is integrated into the existing architecture exploiting the pipelining in order to minimize the latency overhead. When all the words are fetched into the PE, prefetching continues to load both residue codes. Control signals are set up so that the residue prediction is an atomic operation, with no impact on or from other operations (carries, overflows, etc.). Input residues are ready as soon as the PE finishes processing the last word. In addition, the residue fits the PE size since it is smaller (in our implementation, it is half the PE word size), therefore the operation can be completed in a single clock cycle. Using only few additional controls, the residue prediction can be achieved with almost no increase in the circuit area and with a single added clock cycle to the overall latency of each operation. If we consider that each operation requires 4 clock cycles per word and any operand is made of several words, the overhead for residue prediction becomes negligible. For example, in

our reference implementation we have a 128-bit-word memory module, a 32-bit PE and an operand size that starts from 768 bits. In this case, the overhead is only 3.7%. With longer operands, the overheads become significantly smaller.

Finally, the resulting residue must be validated against the computed data. This validation does not have to be scheduled immediately. It is possible to delay the validation to a later time, for instance before reading the result from the memory. This is possible since any occurring error in the data does not affect the check bits during computation; on the other hand, local generation of the check bits (i.e., just before any operation) would force to schedule a code validation checkpoint after each operation, in order to avoid residue generation from corrupted data. Our solution follows the former approach. In particular, an error would not be detected at an immediate checkpoint only if the corrupted value had the same check bits as the correct data. This implies that the detection coverage is inversely proportional to the size of the check modulus which can be chosen accordingly. It should be clear that such a fault will not be detected even afterward.

In our implementation, the memory read policy is changed. In the basic architecture, each memory word had to be individually read, by setting up the read address properly and issuing the read command. In the error detecting version, only the initial address must be submitted. Subsequent memory reads are automatically fetched, while an additional buffer intercepts the data coming from the memory and computes the final residue on-the-fly. Finally, the actual check bits are compared with the predicted check bits, which are stored in the last position. If the two match and no errors are detected, then the read process is repeated and the output is enabled, allowing for external reads. Reissuing the read command may seem a waste of time, but the architecture was developed with area constrained implementation as a goal. Using a buffer to store the data when it

Table 1. Synthesis results – Area does *not* include any memory module

Version	Area (GE)	Latency (ns)
Basic	11,400	4.7
Error Detecting	13,400	4.7

Table 2. Area and latency overheads

Key Length *(bits)*	Global Overheads Area	Memory	Latency
768	+17.8%	+14.3%	+3.7%
1024	+17.8%	+11.1%	+2.9%
1536	+17.8%	+7.7%	+2.0%
2048	+17.8%	+5.9%	+1.5%

is read from memory would have resulted in a large area overhead, while the few additional clock cycles are negligible with respect to the complete process.

The check bits validation unit makes reuse of the residue generation unit described above. In principle, a new residue generator could be implemented. However, the area overhead (an additional increase of 10.3%) would not be compensated by a significant reduction in the delay. Both architectures, in fact, were able to run at the target frequency of 200 MHz. The latency overheads were obtained by running several simulations in ModelSim with realistic data,

while the area figures were obtained by synthesizing both designs with Design Compiler by Synopsys with STM 0.18μm High-Speed libraries.

5 Conclusions

One of the most effective techniques for attacking a cryptosystem is through deliberate error injection during computation. The faulty results can be used by attackers to retrieve the secret keys after a few attempts. In this paper, we extend an RSA architecture to include error detection capabilities based on the residue code. Our design incurs a 17.8% overhead in circuit area and only small latency and memory overheads, which become even smaller with longer keys.

The expected fault coverage, based on our previous experience with error detecting codes and on some simulations, depends on the level of redundancy, i.e., the size of the residue base modulus. After injecting an error, the data and the corresponding check bits become uncorrelated: the error is not detected if and only if the check bits match the data by chance. The probability of this event occurring, when $2^h - 1$ is the residue base modulus, is $(2^h - 1)^{-1} \approx 2^{-h}$.

References

1. R. Anderson and M. Kuhn, "Low Cost Attacks on Tamper Resistant Devices," *Proc. of the International Workshop on Security Protocols*, Lecture Notes in Computer Science, Springer-Verlag, 1997.
2. C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, J.-P. Seifert, "Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures," Cryptographic Hardware and Embedded Systems - CHES 2002, *Lecture Notes in Computer Science*, Vol. 2523, pp. 260-275, Springer-Verlag, 2003.
3. E. Biham, A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," Technical Report, Technion - Computer Science Department, 1997.
4. D. Boneh, R. DeMillo, R. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations," *Journal of Cryptology*, vol. 14, pp. 101-119, 2001.
5. S. Gueron, "Fault Detection Mechanism for Smartcards Performing Modular Exponentiation," *Workshop on Fault Diagnosis and Tolerance in Cryptography 2004*, Supplemental Volume of the 2004 Intern. Conf. on Dependable Systems and Networks, pp. 368-372, 2004.
6. A. Mazzeo, L. Romano, G.P. Saggese, N. Mazzocca, "FPGA-based implementation of a serial RSA processor," *Design, Automation and Test in Europe Conference and Exhibition '03*, pp. 582 - 587, 2003.
7. A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.
8. B. Parhami and A. Avizienis, "Design of Fault-Tolerant Associative Processors," ISCA, pp. 141-145, 1973.
9. R. L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, Vol. 21, Issue 2, pp. 120–126, ACM Press, 1978.
10. A. Shamir, "Method and Apparatus for Protecting Public Key Schemes from Timing and Fault Attacks," US Patent 5991415, 1999.

11. C. Walter, "Montgomery's Multiplication Technique: How to Make It Smaller and Faster", Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Proceedings. Lecture Notes in Computer Science, Vol. 1717, pp. 80-93, 1999.
12. C. Walter, "Data Integrity in Hardware for Modular Arithmetic," Workshop on Cryptographic Hardware and Embedded Systems (CHES), Lecture Notes in Computer Science, Vol. 1965, pp. 204-215, 2000.
13. S.-M. Yen, S. Moon, J.-C. Ha, "Hardware Fault Attack on RSA with CRT Revisited," *Information Security and Cryptology - ICISC '02*, Lecture Notes in Computer Science, Vol. 2587, pp. 374-388, 2002.
14. S.-M. Yen, S. Moon, J.-C. Ha, "Permanent Fault Attack on the Parameters of RSA with CRT," Lecture Notes in Computer Science, Vol. 2727, pp. 285-296, 2003.