Montgomery Exponentiation with no Final Subtractions: Improved Results

Gaël Hachez and Jean-Jacques Quisquater

Université Catholique de Louvain, UCL Crypto Group
Place du Levant, 3, B-1348 Louvain-la-Neuve, Belgium
{hachez,quisquater}@dice.ucl.ac.be

Abstract. The Montgomery multiplication is commonly used as the core algorithm for cryptosystems based on modular arithmetic. With the advent of new classes of attacks (timing attacks, power attacks), the implementation of the algorithm should be carefully studied to thwart those attacks. Recently, Colin D. Walter proposed a constant time implementation of this algorithm [17,18]. In this paper, we propose an improved (*faster*) version of this implementation. We also provide figures about the overhead of these versions relatively to a speed optimised version (theoretically and experimentally).

Keywords: Montgomery multiplication, modular exponentiation, smart cards, timing attacks, power attacks.

1 Introduction

In RSA based crypto-systems, modular exponentiations are often computed with Montgomery multiplications [14].The optimisation of this algorithm is consequently very important. Several fast implementations of this algorithm were proposed both in hardware (e.g. [18]) and software (e.g. [10,6]). These implementations were mainly designed to achieve speed gains.

Recently, a new range of attacks (timing attacks [11] and power attacks [12]) appeared. These attacks are based on side-channel information that are leaked by the hardware device. The tricks used to optimise to the utmost the speed of the algorithm usually amplify this side-channel information. Therefore, new implementations of the algorithm are being created to reduce these threats while almost preserving the speed performance.

In two recent papers [17,18], Colin D. Walter shows that, with a correct implementation, it is possible to make a complete exponentiation based on Montgomery multiplications without any modular reduction (even at the end of the exponentiation) [1]. His implementation is slower than an optimised one although a security gain is achieved against timing attacks and power attacks.

[1] Similar results were already obtained for slower modular multiplication algorithms such as Barrett and Quisquater multiplications (see [6]).

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 293–301, 2000.

The author focuses on hardware implementations while neglecting software implementations that are commonly used even in embedded hardware such as smart cards[2].

Here, we will show a tighter bound on the assumptions made by Colin D. Walter that allow us to speed up software implementations. To illustrate this gain, we will show some figures about performance on a 32-bit RISC-based chip for smart card.

In hardware, the situation is more complex. Usually the tighter bound will either speed up a hardware implementation, or reduce the size of the circuitry needed to obtain this implementation of the Montgomery multiplication. In a particular case, if the size of the modulus is smaller than the size of the multiplier, the new implementation is not suitable.

2 Montgomery Multiplication

The Montgomery multiplication is an algorithm used to compute the product of two integers A and B modulo an integer N.

Because A and B are, for security reasons, quite large, the multiplication is computed with A and B decomposed in small blocks. Those blocks usually have a length t of 8, 16, 32, 64 bits and each number can be decomposed in the form $X = \sum_{i=0}^{p-1} x_i 2^{it}$ where p is the number of blocks needed to represent all numbers used in the algorithm.

The Montgomery multiplication algorithm is described in Fig. 1. As Barrett [2,3] and Quisquater [15,16] modular multiplication, this one does not require any division (expensive operation in hardware). Here, the multiplication is done from left (high order bits) to right (low order bits) which is not the classical order used to make a multiplication.

```
{Pre-condition: $N$ prime to $2^t$}
$S = 0$
for $i = 0$ to $p - 1$
    $q_i = (s_0 + a_i b_0) n'_0 \bmod 2^t$
    $S = (S + a_i \times B + q_i \times N) \operatorname{div} 2^t$
    {Invariant: $0 \leq S < N + B$}
endfor
{Post-condition: $S 2^{pt} = A \times B + Q \times N$}
```

Fig. 1. Montgomery Multiplication

The value n'_0 is computed so that $n_0 \times n'_0 \equiv 1 \bmod N$. The integer p must be chosen such that $N < 2^{pt}$. For more details on the algorithm, see [14,6,18].

[2] The latest chip developed by ST Microelectronics, the smartJ 22 contains software implementation of public key primitives.

3 Montgomery-Based Exponentiation

3.1 Description

The Montgomery multiplication is the basic component used to implement a classical square and multiply algorithm that computes an exponentiation. The result of a Montgomery multiplication ($\overline{\times}$) is not $A \times B \bmod N$ but rather $A \times B \times 2^{-pt} \bmod N$. To obtain a correct result at the end of the exponentiation, we need to make a pre-multiplication ($A \;\overline{\times}\; 2^{2pt} \bmod N$) and a post-multiplication ($A^e \;\overline{\times}\; 1 \bmod N$).

With the following assumptions: $A < 2N, t \geq 1$ and $2N < 2^{(p-1)t}$ C. Walter [17,18] proves that the end-result of the exponentiation (E) is lower than the modulus (N) and does not need any further modular reduction. We will rapidly sketch out the proof.

Proof. Because the result of the multiplication is used as input for the next multiplication, the output must have the same bound as the input. At the second last iteration, we have $S' < N + B$. The assumptions $A < 2N$ and $2N < 2^{(p-1)t}$ guarantee that $a_{p-1} = 0$. Therefore at the last iteration, we have $S < N + 2^{-t}B < 2N$.

At the last multiplication of the exponentiation, we have $A^e < 2N$. The post-multiplication by 1 will remove the possible last reduction. We have at the end: $E2^{pt} = A^e + QN$. $Q < 2^{nt}$ and $A^e < 2N$ implies that: $E2^{pt} < (2^{pt} + 1)N$. We obtain $S \leq N$ (S is an integer). The last case $S = N$ is removed because it implies that $A^e \equiv 0 \bmod N$ and therefore $A \equiv 0 \bmod N$. This signifies that either $A = 0$ (no reductions) or $A = N$ (in a classical crypto-system, $A < N$). □

3.2 Shortcomings

The first part of the proof shows the non-growing property of the Montgomery multiplication. With $A, B < 2N, t \geq 1$ and $2N < 2^{(p-1)t}$ the output of the multiplication is bound: $S < 2N$.

While this result is true, we should not forget the pre-multiplication phase. In this pre-multiplication the integer A is multiplied by 2^{2pt} that is obviously greater than $2N$ and thus we have no insurance that S will be bounded by $2N$ after this pre-multiplication. Therefore, we can not be sure that the result at the end of the exponentiation will not require a final reduction.

We have two solutions to avoid that (proposed in [7,8]):

- pre-compute $2^{2pt} \bmod N$
- use a normal modular multiplication algorithm (Barrett or Quisquater) and compute $A \times 2^{pt} \bmod N$.

Besides this little problem, performance is impeded by one assumption. The $2N < 2^{(p-1)t}$ condition can be very annoying. Specially if we take classical sizes for N and t.

Example 1. We have a modulus N (512 bits) and a 32x32 multiplier ($t = 32$), then we need $p = 18$ instead of $p = 16$ which lowers the performance because the number of multiplications is $O(p)$. With non classical sizes of modulus such as 510 bits, we obtain $p = 17$ instead of $p = 16$ which is less annoying.

For the rest of the paper, we will suppose that we are in a typical case where the size of N is equal to 512, 768, 1024, 2048 bits and $t = 32$.

3.3 Bound Optimisation

We can improve this bound and prove that the result ($S < 2N$) still holds even with $N < 2^{(p-1)t}$ and with a tighter constraint on t: that is, $t \geq 2$ which is obviously not a problem in a software implementation.

In hardware, this can be a problem. If the size of N is less than 2^t, this result does not stand. However this situation does not happen very often as, nowadays, the minimum size for N is at least 512 bits.

At each step of the algorithm the following bound is satisfied: $S < N + B$. From $N < 2^{(p-1)t}$ and $A < 2N$, we know that $a_{p-1} \in \{0, 1\}$. If we start from the second last iteration we have that:

$$\begin{aligned}
S' &= (S + a_{p-1} \times B + q_{p-1} \times N) \text{ div } 2^t \\
S' &\leq (S + B + q_{p-1} \times N) \text{ div } 2^t \\
S' &\leq (S + B + (2^t - 1) \times N) \text{ div } 2^t \\
S' &< (N + B + B + (2^t - 1) \times N) \text{ div } 2^t \\
S' &< (2B + 2^t \times N) \text{ div } 2^t \\
S' &< 2B \text{ div } 2^t + N \\
S' &< 4N \text{ div } 2^t + N \\
S' &< 2N \quad \square
\end{aligned}$$

The remaining of the proof is the same as Walter's one because he does not require anymore that $2N < 2^{(p-1)t}$. Therefore, we proved that we still avoid a final reduction at the end of the exponentiation with better bounds.

Example 2. In the previous example, this new bound is $p = 17$ which is worse than the classical algorithm but better than Walter's version.

4 Speed Analysis

4.1 Building a Generic Model

We can build an approximative model of the number of operations required for a Montgomery multiplication. Let C_A represent the number of clock cycles for an addition and C_M the number of clock cycles for a multiplication. At each step, we need:

- $(2C_A + 2C_M)p$ clock cycles for computing S
- $C_A + 2C_M$ clock cycles for computing q_i.

We need to make a final subtraction in the case of the original Montgomery multiplication: this final subtraction takes C_Ap clock cycles. So we have the following formulae to compute the approximative clock cycles required for a Montgomery multiplication:

- $((2C_A + 2C_M)p + C_A + 2C_M)p$,
- $((2C_A + 2C_M)p + 2C_A + 2C_M)p$ with a final subtraction.

4.2 Adaptation to the ARM7M

We already had a cryptographic library that was designed in the European project CASCADE [4] by J.-F. Dhem. The library runs on an ARM7M CPU (this CPU is used in the GemXpresso 2.0 smart card from Gemplus). Therefore, we used this platform to experimentally compare the performance of the implementations.

The ARM7M is a pure RISC processor. It does not hold any division instructions and there is no support for floating point operations. On the ARM7M, an addition takes 1 clock cycle ($C_A = 1$). The multiplication is a little more complex. The ARM7M possess a dedicated multiply unit that is able to multiply 32x8 bits. Therefore, to multiply 32x32 bits and obtain a 64 bits result, this unit must be used four times. If we add the setup time, a multiplication usually takes 6 clock cycles ($C_M = 6$).

The time taken by the multiplication is not always constant due to optimisations in the ARM7M. If one of the 8 bits blocks of the operand is null, this sub-part of the multiplication is skipped. More details are available in [1]. In particular, if the operand is null then the number of clock cycles decreases from 6 to 2 (the setup time only).

Remembering that the block $a_{p-1} \in \{0, 1\}$, if we take one block more, we need to adapt the above formulae to deal with this non-constant time. So if we take one block more (this paper), we consider that the last block's multiplication for computing S takes only 2 clock cycles[3] and if we take two blocks more (Walter's version), we consider that the last two blocks' multiplication takes only two clock cycles. We obtain thus the following estimations in Table 1.

4.3 Speed Comparison

The library we use has been protected against timing attacks. The original version of the Montgomery algorithm always makes a subtraction after the multiplication and chooses to take the result of the subtraction if it is greater than zero, otherwise the result remains unchanged. A modification was made to avoid timing attacks by adding cycles to have the same timing when the result of the subtraction must be discarded. See [5,9] for timing attacks on this library.

[3] This is a valid approximation because most of the time $a_{p-1} = 0$

Table 1. Formulae (based on a simple model of the ARM7) used to predict the number of clock cycles required for the different versions of the algorithm.

Value	This paper	Walter's version
q_i	$C_A + 2C_M$	$C_A + 2C_M$
S	$(2C_A + 2C_M)p + 2C_A + 2C_{M'}$	$(2C_A + 2C_M)p + 2(2C_A + 2C_{M'})$

Table 2. Predicted time increase for a multiplication ($C_A = 1$, $C_M = 6$) relatively to the standard version with an ending modular reduction ($(14p + 14)p$).

Size of N	This paper	Walter's version
	$(14p + 6 + 13)(p + 1)$	$(14p + 12 + 13)(p + 2)$
512 bits ($p = 16$)	8.5 %	17.7 %
768 bits ($p = 24$)	5.6 %	11.7 %
1024 bits ($p = 32$)	4.2 %	8.8 %
2048 bits ($p = 64$)	2.1 %	4.4 %

However because those added cycles come from an empty loop, this is not a protection against power attacks [13,12].

If we compare predicted results in Table 2 and real results in Table 3, we can see some divergence. This is normal due to the following facts:

- The prediction is made on one multiplication and we get the results on a complete exponentiation without taking the added time into account.
- There is a 3-stage pipeline in the ARM7.
- This is a basic model (no memory operations are taken into account).

It is crucial to note the improvement will be far higher if we take a CPU architecture where the multiplication takes a constant time whatever the value of the operands. Suppose that the time of a multiplication is the same as the time of the addition and equals one clock cycle, we obtain the following results in Table 4.

5 Security Considerations

Today, in smart cards, absolute performance is not the only objective for algorithms anymore. New kinds of side channels based attacks (like the time [11], the power [12]) appeared and security algorithms must be protected against them. This is usually done at the expense of the performance of algorithms. We will see how this algorithm theoretically performs against timing and power attacks.

Table 3. Average time increase for an exponentiation relatively to the standard version with an ending modular reduction.

Size of N	This paper	Walter's version
512 bits	6.3 %	17.6 %
768 bits	4.3 %	11.9 %
1024 bits	3.3 %	9 %
2048 bits	1.6 %	4.5 %

Table 4. Predicted time increase for a multiplication $(C_A, C_M = 1)$ relatively to the standard version with an ending modular reduction $((4p+4)p)$.

Size of N	This paper	Walter's version
	$(4(p+1)+3)(p+1)$	$(4(p+2)+3)(p+2)$
512 bits $(p = 16)$	10.9 %	24 %
768 bits $(p = 24)$	7.3 %	15.9 %
1024 bits $(p = 32)$	5.5 %	11.9 %
2048 bits $(p = 64)$	2.7 %	5.9 %

5.1 Timing Attacks

The original speed optimised algorithm is already protected against timing attacks. Against such attacks our version does not add more security. However this is a cleaner design than always perform a subtraction and add an empty loop (if needed) at the end of the exponentiation.

5.2 Power Attacks

In the original speed optimised version, after the always performed final subtraction, a conditional instruction must decide whether the result of the final subtraction must discarded. Because the result is returned by value and not by address, if the result must be kept, it must be copied. To avoid timing attacks, in the other case (no copy), an empty loop is executed to simulate the time taken by the copy. This method can be easily detected in a power attack. In our new version, a security gain is achieved because no conditional instructions exist anymore.

At first sight, it can only be considered as a security gain because it will not be sufficient to protect against power attacks. Indeed, attacks can be mounted on the exponentiation algorithm independently of the multiplication algorithm as, here, a conditional Montgomery multiplication is executed within the exponentiation algorithm depending on the value of each key bit. This is unrelated to the multiplication algorithm used, it depends on the exponentiation algorithm (attacks of this type were done in [13]).

6 Conclusion

We notice an important improvement of the performance with this version of the Montgomery multiplication but it remains slower than the speed optimised version. With a more generic platform than the ARM7, we should obtain even better improvements as shown in Table 4.

The security gain is related to power attacks [12] against smart cards as there are no more conditional reductions. However, this is not sufficient because the exponentiation algorithm itself is not protected against power attacks.

References

1. ARM. *ARM 7TDMI Data Sheet*, August 1995. Document number: ARM DDI 0029E.
2. P. Barrett. Communications, Authentication and Security Using Public Key Encryption - A Design for Implementation. Master's thesis, Oxford University, September 1984.
3. P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In A. M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86*, volume 263 of *LNCS*, pages 311–323. Springer-Verlag, 1987.
4. CASCADE (Chip Architecture for Smart CArds and portable intelligent DEvices). `http://www.dice.ucl.ac.be/crypto/cascade/`, 1997.
5. J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems. A Practical Implementation of the Timing Attack. In *CARDIS '98*, LNCS. Springer-Verlag, 1998. to appear.
6. Jean-François Dhem. *Design of an Efficient Public-key Cryptographic Library for RISC-based Smart Cards.* Ph.D. Thesis, Université Catholique de Louvain, May 1998.
7. Stephen E. Eldridge. A Faster Modular Multiplication Algorithm. *Inter. J. Comput. Math.*, 40:63–68, 1991.
8. Stephen E. Eldridge and Colin D. Walter. Hardware Implementation of Montgomery's Modular Multiplication Algorithm. *IEEE Transactions on Computers*, 42(6):693–699, June 1993.
9. Gael Hachez, François Koeune, and Jean-Jacques Quisquater. Timing Attack: What Can Be Achieved by a Powerful Adversary? In A. Barbé, E.C. van der Meulen, and P. Vanroose, editors, *The 20th symposium on Information Theory in the Benelux*, pages 63–70, May 1999.
10. Kouichi Itoh, Masahiko Takenaka, Naoya Torii, Syouji Temma, and Yasushi Kurihara. Fast Implementation of Public-Key Cryptography on a DSP TMS320C6201. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES '99*, volume 1717 of *LNCS*, pages 61–72. Springer-Verlag, August 1999.
11. Paul Kocher. Timing Attack on Implementations of Diffie-Hellman, RSA, DSS and other systems. In Neil Kobliz, editor, *Advances in Cryptology - CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, August 1996.
12. Paul Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer-Verlag, August 1999.

13. Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Power analysis Attack of Modular Exponentiation in Smartcards. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES '99*, volume 1717 of *LNCS*, pages 144–157. Springer-Verlag, August 1999.
14. Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.
15. Jean-Jacques Quisquater. Procédé de Codage selon la Méthode dite RSA, par un Microcontrôleur et Dispositifs Utilisant ce Procédé. Demande de brevet français. (Dépôt numéro: 90 02274), February 1990.
16. Jean-Jacques Quisquater. Encoding System According to the So-called RSA Method, by Means of a Microcontroller and Arrangement Implementing this System. U.S. Patent 5,166,978, November 1992.
17. Colin D. Walter. Montgomery Exponentiation Needs no Final Subtractions. *Electronics Letters*, 35(21):1831–1832, October 1999.
18. Colin D. Walter. Montgomery's Multiplication Technique: How to Make It Smaller and Faster. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES '99*, volume 1717 of *LNCS*, pages 80–93. Springer-Verlag, August 1999.