# Optimal Use of Montgomery Multiplication on Smart Cards

Arnaud Boscher and Robert Naciri

Oberthur Card Systems SA,
71-73, rue des Hautes Pâtures,
92726 Nanterre Cedex, France
{a.boscher,r.naciri}@oberthurcs.com

**Abstract.** Montgomery multiplication is used to speed up modular multiplications involved in public-key cryptosystems. However, it requires conversion of parameters into $N$-residue representation. These additional pre-computations can be costly for low resource devices like smart cards. In this paper, we propose a new, more efficient method, suitable for smart card implementations of most of public-key cryptosystems. Our approach essentially consists in modifying the representation of the key and the algorithm embedded in smart card in order to take advantage of the Montgomery multiplication properties.

**Keywords**: Montgomery Multiplication, Smart Card, RSA, ECDSA, GQ2.

## 1 Introduction

Almost all public-key cryptosystems embedded in low resource devices, such as smart cards and PDAs, require an efficient implementation of modular multiplication.

One of the best methods of modular multiplication is due to P.L. Montgomery [1]. It consists in replacing division by an arbitrary number with division by a fixed-number, which can be chosen to be a power of 2 for efficiency reasons. Montgomery multiplication requires pre-computation of a constant to change the representation of the operands. This pre-computation requires time and memory space and must be performed each time the cryptosystem is computed. We will see how most of the public-key cryptosystems can be implemented on a smart card using Montgomery multiplication without this drawback.

The paper is organized as follows. In Section 2, we recall the basics about Montgomery multiplication. In Sections 3 and 4 we propose a method for RSA and CRT RSA implementations using Montgomery multiplication. In Section 5, we adapt the method to GQ2 algorithm [6] which results in an improvement of up to 50 % in execution time compared to the classical methods. Lastly, we look at ECDSA signature [7] in Section 6.

## 2 Montgomery Multiplication

Throughout the rest of the paper, we use $\cdot$ to denote classical multiplication and $*$ to denote Montgomery multiplication.

Let $b$ be the length of the machine word (typically $b = 2^k$ with $k = 8, 16$ or 32). Let $X, Y$ and $N$ be three integers of length $n : X = (x_{n-1}...x_0)_b$, $Y = (y_{n-1}...y_0)_b$. We denote by $R$ the value $b^n$.

For $N$ odd, the Montgomery multiplication of $X$ and $Y$ modulo $N$ is defined by:

$$X * Y \bmod N = X \cdot Y \cdot R^{-1} \bmod N \ .$$

It can be computed by applying the following algorithm shown in [3]:

**Algorithm 2.1** Montgomery multiplication
INPUT: $X, Y, N, R$ and $N' = -N^{-1} \bmod b$
OUTPUT: $X \cdot Y \cdot R^{-1} \bmod N$

1. $A \leftarrow 0$
2. For $i$ from 0 to $n-1$ do
   (a) $u \leftarrow (a_0 + x_i \cdot y_0)N' \bmod b$
   (b) $A \leftarrow (A + x_i \cdot y + u \cdot N)/b$
3. If $A \geq N$ then $A \leftarrow A - N$
4. Return($A$)

Let us denote by $^*$ the Montgomery exponentiation defined by:

$$X^{*e} \bmod N = X^e \cdot R^{1-e} \bmod N \ . \tag{1}$$

As it can be deduced from Relation (1), classical modular exponentiation can be computed using Montgomery exponentiation. First, we have to change the representation of the operand, then carry out the Montgomery exponentiation and finally correct its output to obtain the expected result. This can be summarized by:

$$X^e \bmod N = [(X * R^2)^{*e}] * 1 \bmod N \ ,$$

or by the following algorithm taken from [3]:

**Algorithm 2.2** Modular Exponentiation using Montgomery Multiplication
INPUT: $X, e, N, R$
OUTPUT: $X^e \bmod N$

1. $\tilde{X} \leftarrow X * R^2 \bmod N$
2. $A \leftarrow R \bmod N$

3. For $i$ from $n-1$ to 1 do
   (a) $A \leftarrow A * A \bmod N$
   (b) If $e_i = 1$ then $A \leftarrow A * \tilde{X} \bmod N$
4. $A \leftarrow A * 1 \bmod N$
5. Return($A$)

---

The value $\tilde{X} = X \cdot R \bmod N = X * R^2 \bmod N$ is called the Montgomery representation of $X$. To obtain this representation, the value $R^2 \bmod N$ must be computed. In order to do this, one can use Montgomery multiplication and the following proposition:

**Proposition 1.** *Let $R$ and $N$ be two integers with $N$ odd, then we have:*

$$R^2 \bmod N = (2 \cdot R)^{*log_2[R]} \bmod N \ .$$

*Proof.*

$$\begin{aligned}(2 \cdot R)^{*log_2[R]} &= 2^{log_2[R]} \cdot R^{log_2[R]} \cdot R^{1-log_2[R]} \bmod N \\ &= 2^{log_2[R]} \cdot R \bmod N \\ &= R \cdot R \bmod N \\ &= R^2 \bmod N \ .\end{aligned}$$

□

As a consequence of Proposition 1, the pre-computation of $R^2 \bmod N$ requires $log_2[R]$ Montgomery multiplications. As $R$ equals $b^n$, for public-key cryptosystems using large parameters $n$ (like RSA), this can be a problem in terms of time or memory on low cost devices.

On smart cards for instance, initialization of the parameters can take more time than the Montgomery multiplication itself. One reason is that initialization is made by software, whereas Montgomery multiplication is made by hardware. Another reason is the clock frequency: dedicated hardware for Montgomery multiplication has higher clock frequency than classical CPU.

In the next section, we introduce a new method of computing RSA signatures with Montgomery multiplications without the pre-computation of $R^2 \bmod N$.

## 3 RSA

### 3.1 Classical Method for RSA

The RSA cryptosystem [4] uses a public modulus $N$, product of two large prime numbers $p$ and $q$, a public exponent $e$ co-prime with $\phi(N) = (p-1) \cdot (q-1)$ and a private exponent $d$, inverse of $e$ modulo $\phi(N)$.

To sign a message $M$ using Montgomery multiplication, one can apply the following algorithm:

**Algorithm 3.1** RSA using Montgomery multiplication
INPUT: $M, d, N, R$
OUTPUT: $M^d \bmod N$

1. $X \leftarrow R^2 \bmod N$
2. $\tilde{M} \leftarrow M * X \bmod N$
3. $S \leftarrow \tilde{M}^{*d} \bmod N$
4. $S \leftarrow S * 1 \bmod N$
5. Return($S$)

### 3.2 Our New Method for RSA

Let us assume that the public exponent $e$ is known (as it is often the case). We give in the following a new way of computing a RSA signature:

**Algorithm 3.2** Optimized RSA using Montgomery multiplication
INPUT: $M, d, e, N$
OUTPUT: $M^d \bmod N$

1. $S \leftarrow 1^{*(e-1)} \bmod N$
2. $S \leftarrow M * S \bmod N$
3. $S \leftarrow S^{*d} \bmod N$
4. Return($S$)

Before arguing the correctness of Algorithm 3.2, let us notice that:

$$1^{*(e-1)} \bmod N = 1^{e-1} \cdot R^{1-e+1} \bmod N = R^{2-e} \bmod N \ .$$

So, after the first step of Algorithm 3.2 we have:

$$S = R^{2-e} \bmod N \ .$$

And from the second step, we obtain:

$$\begin{aligned} S = M * S \bmod N &= M * R^{2-e} \bmod N \\ &= M \cdot R^{2-e} \cdot R^{-1} \bmod N \\ &= M \cdot R^{1-e} \bmod N \ . \end{aligned}$$

Finally, using Fermat's little theorem in the last step:

$$\begin{aligned} S = S^{*d} \bmod N &= (M \cdot R^{1-e})^{*d} \bmod N \\ &= M^d \cdot R^{(1-e)d} \cdot R^{1-d} \bmod N \\ &= M^d \cdot R^{1-ed} \bmod N \\ &= M^d \bmod N \ . \end{aligned}$$

Algorithm 3.2 works for every $e$, but is especially interesting when $e$ is small (typically $2^{16}+1$).

Even if the total of Montgomery multiplications in Algorithm 3.1 and in Algorithm 3.2 is not very different, the execution time of the second one will be faster in a smart card context. Indeed, the initialization step of operands takes more time than the Montgomery multiplication itself. This is a consequence of the smart card architecture, where Montgomery multiplication uses dedicated hardware.

# 4 CRT RSA

## 4.1 Traditional Method for CRT RSA

When the values $p$ and $q$ are available, one usually applies the Chinese Remainder Theorem and the Garner's algorithm [5] to improve performance of RSA signature. In the so-called CRT mode, RSA involves the 5 parameters $p, q, d_p, d_q, A$, where $d_p = d \bmod p-1, d_q = d \bmod q-1$ and $A = p^{-1} \bmod q$.

The CRT RSA signature of a message $M$ using Montgomery multiplication is given by:

**Algorithm 4.1** CRT RSA using Montgomery multiplication
INPUT: $M, p, q, d_p, d_q, A, R$
OUTPUT: $M^d \bmod N$

1. $X \leftarrow R^2 \bmod p$
2. $\tilde{M} \leftarrow M * X \bmod p$
3. $\tilde{S_p} \leftarrow \tilde{M}^{*d_p} \bmod p$
4. $S_p \leftarrow \tilde{S_p} * 1 \bmod p$
5. $X \leftarrow R^2 \bmod q$
6. $\tilde{M} \leftarrow M * X \bmod q$
7. $\tilde{S_q} \leftarrow \tilde{M}^{*d_q} \bmod q$
8. $S_q \leftarrow \tilde{S_q} * 1 \bmod q$
9. $X \leftarrow R^2 \bmod p$
10. $\tilde{A} \leftarrow A * X \bmod p$
11. $S \leftarrow [(S_q - S_p) * \tilde{A} \bmod p\ ] \cdot p + S_p$
12. Return$(S)$

## 4.2 Our New Method for CRT RSA

We assume that the public exponent $e$ is available. Moreover, we recall that every message $M$ can be written $M_1 \cdot R + M_0$.

If we store in the smart card the value $\tilde{A}$, instead of $A$ itself, we obtain an optimized CRT RSA implementation using Montgomery multiplication:

**Algorithm 4.2** Optimized CRT RSA using Montgomery multiplication
INPUT: $M, p, q, d_p, d_q, \tilde{A}, e$
OUTPUT: $M^d \bmod N$

1. $X \leftarrow 1^{*(e-2)} \bmod p$
2. $S_p \leftarrow (M_1 + M_0 * 1) \bmod p$
3. $S_p \leftarrow S_p * X \bmod p$
4. $S_p \leftarrow S_p^{*d_p} \bmod p$
5. $X \leftarrow 1^{*(e-2)} \bmod q$
6. $S_q \leftarrow ((M_1 + M_0 * 1) \bmod q$
7. $S_q \leftarrow S_q * X \bmod q$
8. $S_q \leftarrow S_q^{*d_q} \bmod q$
9. $S \leftarrow [(S_q - S_p) * \tilde{A} \bmod p\ ].p + S_p$
10. Return $S$

---

After the first step of the algorithm, we have:

$$X = 1^{*(e-2)} \bmod p = R^{3-e} \bmod p \ .$$

Then, the second step gives:

$$S_p = M_1 + M_0 * 1 \bmod p = M_1 + M_0 \cdot R^{-1} \bmod p \ .$$

Hence, at the third step we have:

$$\begin{aligned} S_p * X \bmod p &= S_p * R^{3-e} \bmod p \\ &= (M_1 + M_0 \cdot R^{-1}) \cdot R^{3-e} \cdot R^{-1} \bmod p \\ &= (M_1 + M_0 \cdot R^{-1}) \cdot R^{2-e} \bmod p \\ &= (M_1 \cdot R + M_0) \cdot R^{1-e} \bmod p \\ &= M \cdot R^{1-e} \bmod p \ . \end{aligned}$$

Thus, Montgomery exponentiation (step 4) gives:

$$\begin{aligned} S_p^{*d_p} \bmod p &= (M \cdot R^{1-e})^{*d_p} \bmod p \\ &= (M \cdot R^{1-e})^{d_p} \cdot R^{1-d_p} \bmod p \\ &= M^{d_p} \cdot R^{(1-e)d_p} \cdot R^{1-d_p} \bmod p \\ &= M^{d_p} \cdot R^{1-ed_p} \bmod p \\ &= M^{d_p} \bmod p \ . \end{aligned}$$

For the same reason, we have:

$$S_q^{*d_q} \bmod q = ((M_1 + M_0 * 1) * 1^{(e-2)})^{*d_q} \bmod q.$$

By definition of $\tilde{A}$, we obtain a correct CRT RSA signature.

This CRT RSA implementation using Montgomery multiplication is optimized for smart cards.

# 5 GQ2

## 5.1 Description

GQ2 [6] is a zero-knowledge algorithm whose security is equivalent to the factorization problem. It can be converted to a signature scheme.

Like RSA, GQ2 uses a public modulus $N$, product of two large primes $p$ and $q$. The parameters of the public key are $N$ and two small numbers, $g_1 = 3$ and $g_2 = 5$. The parameters of the private key are two numbers $Q_1$ and $Q_2$ (lower than $N$), verifying the formula: $Q_i^{512} \cdot g_i^2 = 1 \bmod N$ .

Let us recall in the following the GQ2 authentication protocol.

**Algorithm 5.1** GQ2 authentication of a prover by a verifier
INPUT: $N, Q_1, Q_2$
OUTPUT: Success or failure

1. The prover generates a random number $r$ and sends the commitment $W = r^{512} \bmod N$ to the verifier.
2. The verifier sends a 2-byte challenge $d = d_1||d_2$.
3. The prover computes the response $D = r \cdot Q_1^{d_1} \cdot Q_2^{d_2} \bmod N$.
4. The verifier computes $W' = D^{512} \cdot g_1^{2d_1} \cdot g_2^{2d_2} \bmod N$.
5. The verifier returns "Success" if $W' = W$, "Failure" otherwise.

The GQ2 protocol is faster than RSA due to the small length (2×8 bits) of the exponents involved in modular exponentiation. That is why, if Montgomery multiplication is used, computation of the value $R^2 \bmod N$ is very inconvenient: a big part of execution time of the algorithm will be employed for this.

## 5.2 Our New Method for GQ2

To optimize GQ2 algorithm, we propose to store the values $\tilde{Q_1}$ and $\tilde{Q_2}$ in the non-volatile memory of the smart card. This can be performed once, during personalization step of the card, in factory.

The modified GQ2 algorithm executed by the card is:

**Algorithm 5.2** GQ2 authentication with Montgomery multiplication
INPUT: $N, \tilde{Q_1}, \tilde{Q_2}$
OUTPUT: Success or failure

1. The prover generates a random number $r$ and sends the commitment $W = r^{*512} * 1 \bmod N$ to the verifier.
2. The verifier sends a 2-byte challenge d=$d_1||d_2$.
3. The prover computes the response $D = r * \tilde{Q_1}^{*d_1} * \tilde{Q_2}^{*d_2} * 1 \bmod N$.
4. The verifier computes $W' = D^{512} \cdot g_1^{2d_1} \cdot g_2^{2d_2} \bmod N$.

5. The verifier returns "Success" if $W' = W$, "Failure" otherwise.

---

The computed commitment is equal to:

$$\begin{aligned} W &= r^{*512} * 1 \bmod N \\ &= r^{512} \cdot R^{1-512} * 1 \bmod N \\ &= r^{512} \cdot R^{1-512} \cdot R^{-1} \bmod N \\ &= r^{512} \cdot R^{-512} \bmod N \\ &= (r \cdot R^{-1})^{512} \bmod N \ . \end{aligned}$$

So the random used during the rest of the algorithm is $r \cdot R^{-1} \bmod N$.

The computed response is equal to:

$$\begin{aligned} D &= r * \tilde{Q_1}^{*d_1} * \tilde{Q_2}^{*d_2} * 1 \bmod N \\ &= r \cdot \tilde{Q_1}^{*d_1} \cdot \tilde{Q_2}^{*d_2} \cdot R^{-3} \bmod N \\ &= r \cdot \tilde{Q_1}^{d_1} \cdot \tilde{Q_2}^{d_2} \cdot R^{-3} \cdot R^{1-d_1+1-d_2} \bmod N \\ &= r \cdot \tilde{Q_1}^{d_1} \cdot \tilde{Q_2}^{d_2} \cdot R^{-1-d_1-d_2} \bmod N \\ &= r \cdot (Q_1.R)^{d_1} \cdot (Q_2 \cdot R)^{d_2} \cdot R^{-1-d_1-d_2} \bmod N \\ &= r \cdot Q_1^{d_1} \cdot R^{d_1} \cdot Q_2^{d_2} \cdot R^{d_2} \cdot R^{-1-d_1-d_2} \bmod N \\ &= r \cdot Q_1^{d_1} \cdot Q_2^{d_2} \cdot R^{-1} \bmod N \\ &= r \cdot R^{-1} \cdot Q_1^{d_1} \cdot Q_2^{d_2} \bmod N \ . \end{aligned}$$

So the response is valid according to the random used by the card.

This method allows a big improvement compared to the classical method. For example, if the bit-length of the modulus $N$ is 1024, computation of the value $R^2 \bmod N$ requires 10 Montgomery multiplications whereas the computation of $D$ involves between 16 and 32 Montgomery multiplications. So this method for GQ2 algorithm decreases execution time of more than 50% compared to classsical use of Montgomery multiplication.

## 6 ECDSA Signature

### 6.1 Description

Elliptic Curves Digital Signature Algorithm [7] produces short signatures and so, are suitable for smart card. The precedent technique can still be applied in order to improve the time of calculation.

In the following, we only consider the case of elliptic curves over prime fields. Let $(E)$ be the elliptic curve over a finite field of prime characteristic $p$ defined by:

$$y^2 = x^3 + ax + b \ \ with \ a, b \in GF(p) \ \ .$$

Let $G = (x_G, y_G)$ be a point of $(E)$ of order $n$ prime. The ECDSA private key is an integer $d$ such that $d \in [0, n-1]$. The corresponding public key is the point $Q = (x_Q, y_Q) = d \times G$.

The ECDSA signature algorithm is:

**Algorithm 6.1** ECDSA signature
INPUT: $M, (E), G, d, n$
OUTPUT: $r, s$

1. Generate a random number $k$, such that $k \in [1, n-1]$.
2. Compute the elliptic curve point $k \times G = (x_k, y_k)$.
3. Set $r = x_k \bmod n$.
4. Compute $s = k^{-1}(\text{SHA-1}(M) + d \cdot r) \bmod n$.
5. Return$(r, s)$.

## 6.2 First Method

ECDSA involves modular computation over $GF(p)$ for computation of the scalar multiplication described in step 2 of algorithm, but computation over $GF(n)$ for the rest of the algorithm. For clarity reasons, when Montgomery multiplications are executed modulo $p$ (resp. $n$), we use notations $*_p$ (resp. $*_n$) and $R_p$ (resp. $R_n$).

Let $(\tilde{E})$ be the image of $(E)$ using Montgomery representation. It is defined by:

$$\tilde{y}^2 = \tilde{x}^3 + (a.R_p) *_p \tilde{x} + (b.R_p) \ \ .$$

To configure the smart card for ECDSA signature scheme, we need to replace $G = (x_G, y_G)$ by $\tilde{G} = (\tilde{x}_G, \tilde{y}_G) = (x_G.R_p \bmod p, y_G.R_p \bmod p)$ and $d$ by $\tilde{d} = d \cdot R_n \bmod N$ . This rewritten in Montgomery representation is performed once, on a computer, and the modified parameters are stored in the smart card during the personalization phase.

The new ECDSA signature scheme using Montgomery arithmetic is the following:

**Algorithm 6.2** ECDSA signature using Montgomery multiplication
INPUT: $M, (\tilde{E}), \tilde{G}, \tilde{d}, n$
OUTPUT: $r, s$

1. Generate a random number $k$, such that $k \in [1, n-1]$.

2. Compute $k_1 = k *_n 1$.
3. Compute the elliptic curve point $k \times \tilde{G} = (\tilde{x_k}, \tilde{y_k}) = (x_k \cdot R_p \bmod p, y_k \cdot R_p \bmod p)$.
4. Compute $r = \tilde{x_k} *_p 1$.
5. Compute $r = r \bmod n$.
6. Compute $s = k_1^{*(-1)} *_n (\text{SHA-1}(M) + \tilde{d} *_n r) \bmod n$.
7. Return$(r, s)$.

---

This algorithm computes a correct ECDSA signature of message $M$ using only Montgomery multiplications. The correctness of the computation is due to:

$$\begin{aligned}
s &= k_1^{*(-1)} *_n (\texttt{SHA-1(M)} + \tilde{d} *_n r) \bmod n \\
&= k_1^{-1} \cdot R_n^2 *_n (\texttt{SHA-1(M)} + \tilde{d} *_n r) \bmod n \\
&= k_1^{-1} \cdot R_n \cdot (\texttt{SHA-1(M)} + \tilde{d} *_n r) \bmod n \\
&= (k \cdot R_n)^{-1} \cdot R_n \cdot (\texttt{SHA-1(M)} + \tilde{d} *_n r) \bmod n \\
&= k^{-1} \cdot (\texttt{SHA-1(M)} + \tilde{d} *_n r) \bmod n \\
&= k^{-1} \cdot (\texttt{SHA-1(M)} + \tilde{d} \cdot r \cdot R_n^{-1}) \bmod n \\
&= k^{-1} \cdot (\texttt{SHA-1(M)} + d \cdot R_n \cdot r \cdot R_n^{-1}) \bmod n \\
&= k^{-1} \cdot (\texttt{SHA-1(M)} + d \cdot r) \bmod n \ .
\end{aligned}$$

The value $r$ satisfies the following equalities:

$$\begin{aligned}
r &= \tilde{x_k} *_p 1 \bmod p \\
&= \tilde{x_k} \cdot R_p^{-1} \bmod p \\
&= x_k \cdot R_p \cdot R_p^{-1} \bmod p \\
&= x_k \ .
\end{aligned}$$

### 6.3 Second Method

Algorithm 6.2 can also be computed by using the notion of Montgomery inverse introduced by B. Kaliski. The Montgomery inverse of an element $a$ is defined by:

$$a \rightarrow \widehat{a}^{-1} = a^{-1} \cdot R_n \bmod n.$$

B. Kaliski proposed an efficient binary algorithm [8] to compute this inverse.

Using this algorithm and the parameters $(\tilde{E}), \tilde{G}$ and $\tilde{d}$, the ECDSA signature scheme can be optimized for Montgomery multiplication in the following way:

---

**Algorithm 6.3** ECDSA signature using Montgomery multiplication and Kaliski inverse

---

INPUT: $M, (\tilde{E}), \tilde{G}, \tilde{d}, n$
OUTPUT: $r, s$

---

1. Generate a random number $k$, such that $k \in [1, n-1]$.

2. Compute the elliptic curve point $k \times \tilde{G} = (\tilde{x_k}, \tilde{y_k}) = (x_k \cdot R_p \bmod p, y_k \cdot R_p \bmod p)$.
3. Compute $r = \tilde{x_k} *_p 1$.
4. Compute $r = r \bmod n$.
5. Compute $s = \widehat{k}^{-1} *_n (\text{SHA-1}(M) + \tilde{d} *_n r) \bmod n$.
6. Return$(r, s)$.

---

This algorithm computes a correct ECDSA signature of a message $M$ using only Montgomery multiplications. The correctness of the computation is due to:

$$\begin{aligned} s &= \widehat{k}^{-1} *_n (\texttt{SHA-1(M)} + \tilde{d} *_n r) \bmod n \\ &= k^{-1} \cdot R_n *_n (\texttt{SHA-1(M)} + \tilde{d} *_n r) \bmod n \\ &= k^{-1} \cdot R_n . R_n^{-1} (\texttt{SHA-1(M)} + \tilde{d} *_n r) \bmod n \\ &= k^{-1} (\texttt{SHA-1(M)} + d \cdot R_n *_n r) \bmod n \\ &= k^{-1} (\texttt{SHA-1(M)} + d \cdot R_n \cdot r \cdot R_n^{-1}) \bmod n \\ &= k^{-1} (\texttt{SHA-1(M)} + d \cdot r) \bmod n \ . \end{aligned}$$

## 7 Conclusion

We have proposed new ways of using Montgomery multiplication to improve the performance of cryptographic algorithms when they have to be implemented on smart cards.

Our approach comprises two interlocking parts. The first part uses a Montgomery representation to store the private parameters in the smart card. This representation can be computed externally, during the personalization phase of the card, where resource limitations are not a problem. The second part modifies the cryptographic algorithms in order to use a Montgomery representation of the private parameters. This method improves the execution time of the underlying algorithm. For example, a GQ2 authentication is twice as fast compared to the traditional approach. The method is different from those proposed in [9] because the result returned by the card is correct without modifying the protocols. The verifier doesn't need to know how the computation was made.

We have seen that this method can be applied to RSA, GQ2 and ECDSA signature, but it can also be applied for others public-key crypto-systems like ECDSA verification or Feige-Fiat-Shamir [10] for example.

**Acknowledgements**. We would like to thank Emmanuel Prouff for many fruitful comments.

## References

1. P.L. Montgomery. *Modular multiplication without trial division.* Mathematics of computation 44, 1985.

2. D.E. Knuth. *The Art of Computer Programming, vol.2 : Seminumerical Algorithms*. 3rd ed., Addison-Wesley, Reading MA, 1999.
3. A.J. Menezes and P.C. van Oorschot and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
4. R. Rivest, A. Shamir, L. Adleman. *A method for obtaining digital signatures and public-key cryptosystems*. Comm. of the ACM 21: 120-126, 1978.
5. C. Couvreur, J-J. Quisquater. *Fast decipherement algorithm for RSA public-key cryptosystem*. Electronic Letters 18(21): 905-907, 1982.
6. L.C. Guillou, M. Ugon, J-J. Quisquater. *Cryptographic authentication protocols for smart card*. Computer Networks: 437-451, 2001.
7. ANSI X9.62. *Public key cryptography for the financial services industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. 1999.
8. B. Kaliski. *The Montgomery Inverse and its application*. IEEE Transactions on Computers, 44: 1064, 1995.
9. D. Naccache, D. M'Raihi. *Montgomery-Suitable Cryptosystems*. Algebraic Coding 781: 75-81, 1994.
10. U. Feige, A. Fiat, A. Shamir. *Zero-knowledge proofs of identity*. Journal of Cryptology, 1: 77-94, 1988.
11. H. Handschuh, P. Paillier. *Smart Card Crypto-Coprocessors for Public-Key Cryptography*. CryptoBytes 4(1): 6-11, 1998.