Unbridle the Bit-Length of a Crypto-coprocessor with Montgomery Multiplication

Masayuki Yoshino, Katsuyuki Okeya, and Camille Vuillaume

Hitachi, Ltd., Systems Development Laboratory, Kawasaki, Japan
{m-yoshi,ka-okeya,camille}@sdl.hitachi.co.jp

Abstract. We present a novel approach for computing $2n$-bit Montgomery multiplications with n-bit hardware Montgomery multipliers. Smartcards are usually equipped with such hardware Montgomery multipliers; however, due to progresses in factoring algorithms, the recommended bit length of public-key schemes such as RSA is steadily increasing, making the hardware quickly obsolete. Thanks to our double-size technique, one can re-use the existing hardware while keeping pace with the latest security requirements. Unlike the other double-size techniques which rely on *classical* n-bit modular multipliers, our idea is tailored to take advantage of n-bit *Montgomery* multipliers. Thus, our technique increases the perenniality of existing products without compromises in terms of security.

Keywords: Montgomery multiplication, RSA, crypto-coprocessor, smartcard.

1 Introduction

The algorithm proposed by Montgomery to calculate modular multiplications [6], usually referred to as "Montgomery multiplication" technique, is extensively used in practical implementations of public-key cryptosystems such as RSA [10]. In particular, Montgomery multiplications are not affected by delays which are commonly introduced by carries in other strategies for computing modular multiplications. As a consequence, Montgomery's approach is very effective for high-performance hardware implementations of modular multiplications. Low-end devices such as smartcards can benefit from crypto-coprocessors implementing Montgomery multiplications [7], which can drastically reduce the time necessary to encrypt and decrypt data, or sign and verify signatures. But such hardware accelerators suffer from an important restriction: their operand size is limited [8]. Now, because of progresses in integer factorization [11], official security institutions are slowly but surely moving their recommendation from 1024-bit to 2048-bit key sizes for RSA; unfortunately, the latter bit length is not supported by many crypto-coprocessors.

This problem has motivated many studies for developing double-size modular multiplication techniques using single-size hardware multipliers. On the one

E. Biham and A.M. Youssef (Eds.): SAC 2006, LNCS 4356, pp. 188–202, 2007.

hand, thanks to the Chinese Remainder Theorem, *private* computations (decryption or signature generation) require only single-size multiplications for computing a double-size decryption or signature generation [9]. On the other hand, in the case of *public* computations, the Chinese Remainder Theorem is of no help, and double-size modular multiplications are needed.

Paillier initiated the work on double-size multiplications [8], and showed how to efficiently compute a kn-bit classical modular multiplication with n-bit classical modular multiplication units. Later, Fischer et al. optimized Paillier's scheme for the $2n$-bit case [2]. Finally, Chevallier-Mames et al., also concentrating on the case of $2n$-bit multiplications, showed further improvements in the general case and when the modulus has a special form [1]. We note a recurring problem in these techniques: they are based on *classical* modular multipliers, and as such, do not concern about taking high-performance of *Montgomery* multipliers which usually equip smartcards. This is a serious limitation of these techniques, which cannot fully take advantage of the available hardware.

In this paper, we propose a technique for computing $2n$-bit *Montgomery* multiplications with n-bit *Montgomery* multiplication units. Firstly, we define the notion of quotients of n-bit Montgomery multiplications; indeed, such quotients are necessary to calculate $2n$-bit Montgomery remainders. We consider two types of settings, and in each case, propose efficient solutions to compute the quotients. In the first settings, we assume that we have to re-use an existing n-bit Montgomery multiplier, and that we cannot modify it. In this case, we show how to emulate the calculation of the quotient in software with two calls to the n-bit Montgomery multiplier. In the second settings, the modification of the hardware Montgomery multiplier is allowed, but still restricted to n-bit operands. We explain how to modify the circuitry with minimal changes in order to calculate the quotients along with their remainders. In addition to Montgomery quotients, our double-size technique also requires a new representation of $2n$-bit integers, tailored for a better use of Montgomery multipliers. Indeed, to satisfy the requirements of the Montgomery multipliers, the moduli must be odd and greater than 2^{n-1}. Our representation does not only achieve this, but also allows an efficient conversion from/to the standard binary representation of integers.

As a result, thanks to our double-size technique, one can compute $2n$-size Montgomery multiplications with available n-bit hardware Montgomery multipliers, allowing the current generation of crypto-coprocessor to survive the shift towards higher security and longer key lengths.

The rest of this paper is organized as follows. In Section 2, we review previous double-size technique based on classical modular multiplications [2,1]. In Section 3, we introduce the Montgomery multiplications and put their limitations in evidence. In Section 4, we explain our idea for computing $2n$-bit Montgomery multiplications with n-bit Montgomery multipliers. Since our technique requires the n-bit quotients of the Montgomery multiplications, in Section 5 we introduce two approaches to calculate these quotients; the first is well-suited for software implementations, and the second is for hardware implementations.

Section 6 shows experimental results and introduces some practical issues. Finally, we conclude with Section 7.

Notation
We let n denote the operand-size of Montgomery/classical modular multiplication units. We also let capital letters: A, B, N and M denote $2n$-bit integers, and small letters denote the others, such as n-bit integers.

2 Known Double Size Techniques

Low-end devices such as smartcards are equipped with crypto-coprocessors to calculate modular multiplications; however, such hardware accelerators have a strict restriction: their operand size is limited. Recently, because of progresses in integer factorization [11], official security institutions are changing their recommended key-length for RSA from 1024-bit to 2048-bit. Now, this problem has motivated many studies for developing double-size modular multiplication using single-size hardware multipliers and only public information [3].

Paillier first initiated the work on double size-multiplications [8], and showed how to compute a kn-bit classical modular multiplication with n-bit classical modular multiplication units and public information. Later, Fischer et al. [2] optimized Paillier et al.'s scheme for the $2n$-bit case. Finally, Chevallier-Mames et al. [1] showed further improvements in the case of $2n$-bit multiplications, too.

This section introduces works about schemes of Fischer et al. and not Chevallier-Mames et al., because in Section 4 we will propose our double size technique which modifies Fischer et al.'s one.

2.1 Fischer et al.'s Schemes

In this subsection, we introduce the work of Fischer et al. [2]: how to compute double-size *classical* modular multiplication with single-size *classical* modular multiplication units. Their double-size technique requires not only remainders but also quotients of n-bit multiplication to build a $2n$-bit remainder.

The equation $xy - q_c w = r_c$ shows the relation between the product of two integers x and y, the modulus w, the quotient q_c and the remainder r_c in the case of classical modular multiplications. The basic idea of classical modular multiplications is to subtract the modulus w from the most significant bit of product xy, q_c times until the product becomes less than modulus w; thus, digits of the product are eliminated from left to right.

Fischer et al. assumed that the instruction MultModDiv is available, where MultModDiv computes the remainder and the quotient of n-bit classical modular multiplications. For n-bit positive integers x, y and w, $\mathsf{MultModDiv}(x, y, w) = (q_c,\ r_c)$ with $q_c = \lfloor (xy)/w \rfloor$ and $r_c = xy \pmod{w}$.

The MultModDiv instruction is a natural extension of the classical modular multiplication. If the classical modular multiplications is implemented in hardware, the MultModDiv instruction can be emulated with two calls to the multiplier, or by changing the hardware of the multiplier only a little.

Algorithm 1. Fischer et al.'s algorithm

INPUT: $A = a_1 2^n + a_0$, $B = b_1 2^n + b_0$, $N = n_1 2^n + n_0$ with $0 \leq A, B < N$, $2^{2n-1} < N < 2^{2n}$;
OUTPUT: $AB \pmod N$;

1. $(q_1, r_1) = \mathsf{MultModDiv}(b_1, 2^n, n_1)$
2. $(q_2, r_2) = \mathsf{MultModDiv}(q_1, n_0, 2^n)$
3. $(q_3, r_3) = \mathsf{MultModDiv}(a_1, r_1 - q_2 + b_0, n_1)$
4. $(q_4, r_4) = \mathsf{MultModDiv}(a_0, b_1, n_1)$
5. $(q_5, r_5) = \mathsf{MultModDiv}(q_3 + q_4, n_0, 2^n)$
6. $(q_6, r_6) = \mathsf{MultModDiv}(a_1, r_2, 2^n)$
7. $(q_7, r_7) = \mathsf{MultModDiv}(a_0, b_0, 2^n)$
8. **Return** $(r_3 + r_4 - q_5 - q_6 + q_7)2^n + (r_7 - r_6 - r_5)$

We introduce an algorithm proposed by Fischer et al. to compute $2n$-bit classical modular multiplication ($AB \mod N$). Given $2n$-bit integers A, B, N, where $0 \leq A, B < N$. First, every $2n$-bit integers are divided into n-bit integers that can be handled by the $\mathsf{MultModDiv}$ instruction. $A = a_1 2^n + a_0$, $B = b_1 2^n + b_0$, $N = n_1 2^n + n_0$. The equation $n_1 2^n \equiv -n_0 \pmod N$ holds thanks to the above transformation. Their proposed algorithm derives from the equation $n_1 2^n \equiv -n_0 \pmod N$, and is described in Algorithm 1.

3 Montgomery Multiplications

Montgomery multiplications, which are based on a technique to calculate modular multiplication proposed by Montgomery [6], are widely used in practical implementations of public-key cryptosystems, such as RSA. Montgomery multiplications are very suitable for high-performance hardware implementations of modular multiplications, which are typically one of the most expensive operations in public-key cryptosystems. Most low-end devices such as smart cards have crypto-coprocessors implementing Montgomery multiplications to encrypt and decrypt data, or sign and verify signatures.

In this section, we introduce Montgomery multiplications and describe the problems that occur when one tries to extend known double size technique to n-bit Montgomery multipliers.

3.1 Montgomery Multiplication Algorithm

The basic idea of Montgomery multiplications is to replace expensive divisions by cheaper multiplications and additions in computations. For n-bit integers x, y, w, $0 \leq x, y < w$, and $\gcd(w, m)=1$, the Montgomery multiplication algorithm outputs the remainder r, $r := xym^{-1} \pmod w$ where m is called *Montgomery constant* and m^{-1} is the inverse of m modulo w. The value $m = 2^n$ is widely

Algorithm 2. Montgomery multiplication algorithm

INPUT: x, y, w with $0 \leq x, y < w, m = 2^n$, $\gcd(w, m) = 1$ and $w' = -w^{-1} \mod 2$;
OUTPUT: r;

1. $r \leftarrow 0$
2. For i from 0 to $(n-1)$ do the following:
   (a) $u_i \leftarrow (r_0 + x_i y_0) w' \mod 2$
   (b) $r \leftarrow (r + 2r_i + u_i w)/2$
3. If $r \geq w$ then $r \leftarrow r - w$
4. **Return** r

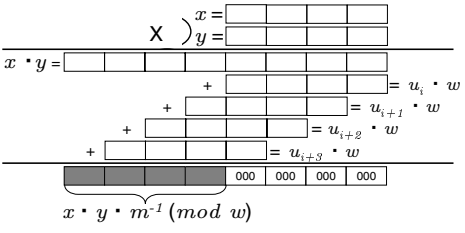

Fig. 1. Basic Idea of Montgomery Multiplication

used in practice because reduction modulo m and division by m are both intrinsically fast operations [5]. Thus the Montgomery algorithm is faster than classical modular multiplication.

Figure 1 illustrates the principle of Montgomery multiplications. Unlike classical modular multiplication techniques, Montgomery multiplications add the modulus w to the product xy from the least significant bit (that is, from right to left), and save the remainder r in the most significant side.

3.2 Problems of Previous Techniques

The schemes proposed by Fischer et al. [2] and Chevallier-Mames et al. [1] show how to compute a $2n$-bit *classical* modular multiplication with n-bit *classical* modular multiplication units. But their schemes cannot take advantage of high-performance *Montgomery* multipliers for the two following reasons:

Problem 1: Quotient
Their double-size techniques require not only n-bit remainders but also n-bit quotients of multiplications to construct $2n$-bit remainders. However, there is no notion of quotient of Montgomery multiplications.

Problem 2: Modulus
The moduli of Montgomery multipliers must be odd because they are restricted to be coprime to Montgomery constants. However, their double-size techniques

allow even moduli. For example, in Algorithm 1, they set upper n-bit value n_1 as modulus, but n_1 can be even.

4 New Double Size Techniques

We propose a new scheme for computing $2n$-bit Montgomery multiplications with the existing coprocessors. On the one hand, Montgomery multiplications are widely implemented on coprocessors for public-key cryptosystems, and produce high-performance hardware modular multiplications for encrypting and decrypting data, or signing and verifying signatures. On the other hand, there is no scheme to compute a double-size Montgomery multiplication efficiently with such coprocessors.

4.1 Instruction for Remainders

First, we define the instruction for computing the remainder of Montgomery multiplications in Definition 1.

Definition 1. *For numbers,* $0 \leq x, y < \min\{w, 2^n\}$, $2^{n-1} < w < 2^{n+1}$, $m = 2^n$, $gcd(m, w) = 1$, *the* MultMon *instruction is defined as* $r = \mathsf{MultMon}(x, y, w)$ *with* $r := xym^{-1} \pmod{w}$.

4.2 Instruction for Quotients

As the schemes of Fischer et al. [2] and Chevallier-Mames et al. [1] require quotients of n-bit classic modular multiplications, our scheme also requires quotients of n-bit Montgomery multiplications to construct $2n$-bit remainder. However, there is no definition of quotients of Montgomery multiplications. To solve this problem, we extend the notion of quotients to the case of Montgomery multiplications.

The remainder calculated by Montgomery multiplications is different from the remainder calculated by classical modular multiplications. Indeed, from Definition 1, the following equation holds; $xy \equiv rm \pmod{w}$, where $m = 2^n$. The above equation means that there is some integer q satisfying: $xy - qw = rm$. We call this integer q *the quotient of the Montgomery multiplication.*

Now, we define the instruction to calculate the quotient and the remainder of the Montgomery multiplication in Definition 2.

Definition 2. *For numbers,* $0 \leq x, y < \min\{w, 2^n\}$, $2^{n-1} < w < 2^{n+1}$, $m = 2^n$, $gcd(m, w)=1$, *the* MultMonDiv *instruction is defined as* $(q, r)= \mathsf{MultMonDiv}(x, y, w)$ *with* $r := xym^{-1} \pmod{w}$ *and* q *satisfies the equation;* $xy = qw + rm$.

In Section 5, we will show an algorithm to implement the MultMonDiv instruction and establish its correctness.

Algorithm 3. Modified Fischer et al.'s Algorithm

INPUT: $A = a_1 z + a_0 m, B = b_1 z + b_0 m, N = n_1 z + n_0 m$, with $M = 2^{2n}$;
OUTPUT: $ABM^{-1} \pmod N$;

1. $(q_1, r_1) = \mathsf{MultMonDiv}(b_1, z, n_1)$
2. $(q_2, r_2) = \mathsf{MultMonDiv}(q_1, n_0, z)$
3. $(q_3, r_3) = \mathsf{MultMonDiv}(a_1, r_1 - q_2 + b_0, n_1)$
4. $(q_4, r_4) = \mathsf{MultMonDiv}(a_0, b_1, n_1)$
5. $(q_5, r_5) = \mathsf{MultMonDiv}(q_3 + q_4, n_0, z)$
6. $(q_6, r_6) = \mathsf{MultMonDiv}(a_1, r_2, z)$
7. $(q_7, r_7) = \mathsf{MultMonDiv}(a_0, b_0, z)$
8. **Return** $(r_3 + r_4 - q_5 - q_6 + q_7)z + (r_7 - r_6 - r_5)m$

4.3 Representation of $2n$-Bit Integers

We define a new representation to divide a $2n$-bit integer into two n-bit integers for n-bit Montgomery multipliers.

Definition 3. *For numbers,* $0 \le A, B < N$, $2^{2n-1} < N < 2^{2n}$, $2^{n-1} < z < 2^n$, *z is odd,* $m = 2^n$ *and* $gcd(m, z) = 1$, *the representation is defined as* $N = n_1 z + n_0 m$, $A = a_1 z + a_0 m$, $B = b_1 z + b_0 m$.

The product $n_0 m$ is always even. Therefore z and n_1 must be odd, whenever N is odd.

4.4 Modified Fischer et al.'s Algorithm

Thanks to Definition 3, we extend schemes of Fischer et al. to the case of Montgomery multiplication in Algorithm 3, which only uses odd moduli n_1 and z. Since n-bit Montgomery multiplications output the n-bit remainder: $xym^{-1} \pmod w$ where $0 \le x, y < w$, $2^{n-1} < w < 2^n$ and $m = 2^n$, our algorithm outputs the $2n$-bit remainder of the Montgomery multiplication: $ABM^{-1} \pmod N$ where $0 \le A, B < N$, $2^{2n-1} < N < 2^{2n}$ and $M = 2^{2n}$.

Theorem 1. *Algorithm 3 computes* $ABM^{(-1)} \pmod N$ *calling length* n-$\mathsf{MultMonDiv}$ *instruction, provided that* $0 \le A, B < N < 2^{2n}$ *and* $M = 2^{2n}$.

We show the proof of Theorem 1 in Appendix A.

5 Implementations for Quotients

In Section 4, we defined the quotient of Montgomery multiplications; in fact, this quotient is necessary to compute $2n$-bit Montgomery multiplications. We consider two types of settings, and in each case, show efficient algorithms to calculate the quotients. In the first settings, we assume that we have to re-use an existing n-bit Montgomery multiplier in software, and cannot modify it. Thus,

Algorithm 4. MultMonDiv instruction calling the MultMon instruction

INPUT: x, y, w with $0 \leq x, y < w$, $2^{n-1} < w < 2^n$, $m = 2^n$ and $\gcd(w, m) = 1$;
OUTPUT: q, r;

1. $r \leftarrow$MultMon(x, y, w)
2. $r' \leftarrow$MultMon$(x, y, w + 2^n)$
3. $tmp \leftarrow xy - rw + r'(w + 2^n) \pmod{2^2}$
4. If $tmp > 2$, then $tmp \leftarrow tmp - 2^2$.
5. $q \leftarrow tmp \times (w + 2^n) + r' - r$
6. **Return** (q, r)

we assume a pure software implementation of our double-size technique. Section 5.1 shows how to emulate the calculation of the quotient with two calls to the n-bit Montgomery multiplier. In the second settings, modifications of the hardware Montgomery multiplier are allowed, but still restricted to n-bit operands. Section 5.2 explains how to modify the circuitry with minimal changes.

5.1 Software Approach: Calling Montgomery Multipliers

In this subsection, we introduce Algorithm 4, which emulates the calculation of quotients with two calls to the n-bit Montgomery multiplier.

Theorem 2. *Algorithm 4 computes* MultMonDiv(x, y, w) *instruction calling length* $(n + 1)$-MultMon *instruction twice, provided that* $0 \leq x, y < w$, $2^{n-1} < w < 2^n$, $m = 2^n$ *and* $gcd(m, w) = 1$.

Appendix B shows the proof of Theorem 2.

5.2 Hardware Approach: Changing Montgomery Multipliers

This subsection shows that the implementation of the MultMon instruction can be changed in order to directly compute the MultMonDiv instruction. In fact, since the MultMon instruction already has information of the quotient, Algorithm 5 has little changes compared to the MultMon instruction, which is the standard technique proposed by Montgomery [6]. We just insert Step 2.(c) to calculate the quotient and output the quotient along with the remainder in Step 4.

6 Experimental Results

6.1 Validation

We implemented 2048-bit Montgomery multiplications and exponentiations on an emulator for smartcards using our proposed technique. Its coprocessor can only handle 1024-bit operands for Montgomery multiplications. Therefore, unlike the assumption in Theorem 2, we make a more strict assumption for getting the quotients: the bit-length of the modulus is exactly twice as much as the operands

Algorithm 5. MultMonDiv instruction with modified MultMon instruction

INPUT: x, y, w with $0 \leq x, y < w, m = 2^n$, $\gcd(w, m) = 1$ and $w' = -w^{(-1)} \mod 2$;
OUTPUT: q, r;

1. $q \leftarrow 0$ and $r \leftarrow 0$
2. For i from 0 to $(n-1)$ do the following:
   (a) $u_i \leftarrow (r_0 + x_i y_0) w' \mod 2$
   (b) $q \leftarrow q + u_i 2^i$
   (c) $r \leftarrow (r + 2r_i + u_i w)/2$
3. If $r \geq w$ then $r \leftarrow r - w$
4. **Return** (q, r)

size of the coprocessor. We show another algorithm in Appendix C for implementing the MultMonDiv instruction with this more strict condition which we faced.

6.2 Practical Implementation Issues

Representations of $2n$-bit Integers
We implemented our technique with w set as $(2^n - 1)$ for two reasons. One reason is that w should be odd because of the requirements of Montgomery multiplications, and that w is $2^{n-1} < w < 2^n$ because of the assumptions in Theorem 2. The other is that the conversion to such representation is easy. We show how to get the representation of $2n$-bit moduli for Montgomery multiplications. $N = n_1' 2^n + n_0' = n_1(2^n - 1) + n_0 2^n$. Then, we can calculate n_1 and n_0 easily for Montgomery multiplications. $n_1 := 2^n - n_0'$ and $n_0 := n_1' - n_1 + 1$.

Condition on the Modulus
Unfortunately, it is not easy to achieve the assumption of Theorem 2, namely that the modulus must be greater than 2^{n-1}. For example, if w is $(2^n - 1)$, $0 < n_1 < 2^n$; n_1 can be smaller than 2^{n-1}. One choice is to modify Algorithm 3 slightly. When the value of n_1 is $0 < n_1 < 2^{n-1}$, $(m - n_1)$ can be applied to Algorithm 3 as modulus instead of n_1 to satisfy the assumption. We can compute the quotient and the remainder of the modulus n_1 from the modulus $(m - n_1)$ with the following equations. $xy = q(m - w) + rm = (-q)w + (q + r)m$.

Handling of Input Value
The input value in Algorithm 3 may break the assumption $(0 < x, y < 2^n)$ of Theorem 2. Since the Montgomery remainder only is affected by this problem, it could be solved by the following fact: if $xy \cdot m^{(-1)} \pmod{w} = r$, then $(x + im)(y + jm) \cdot m^{(-1)} \pmod{w} = r + jx + iy + ijm$ holds, where i and j are small integers.

Reduction of the intermediate output
It often happens that the value of the intermediate output q and r are not reduced. In this case, we have a strategy to compute new integers q' and r'

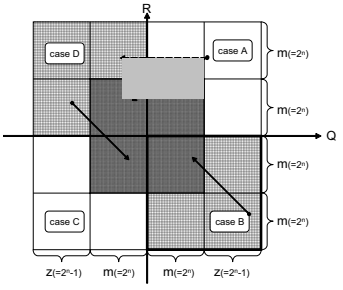

Fig. 2. Reduction for a quotient and a remainder

satisfying the following equation with two vectors; $(z, -m)$ and $(n_0 - z, n_1 + m)$. $(q', r') = (q, r) + i(z, -m) + j(n_0 - z, n_1 + m)$, where i and j are small integers. If one set z as $2^n - 1$ and m as 2^n, the vector $(z, -m)$ is independent of the other $(n_0 - z, n_1 + m)$: The direction of the vector is that $-m/z \approx -1$ and $0 \leq (n_1 + m)/(n_0 - z) \leq 3/4$.

Figure 2 shows works of our strategy to reduce q and r to $|q| < m$ and $|r| < m$ respectively, where $m \leq |q| \leq m + z$, $m \leq |r| \leq m + z$, $m = 2^n$ and $z = 2^n - 1$.

There are four cases to reduce q and r.

Case A: $0 \leq q, r$ and, $m \leq q \leq z + m$ or $m \leq r \leq z + m$.
Compute $(q, r) \leftarrow (q - (n_0 - z), r - (n_1 + m))$, then return (q, r) or go to case A or D.

Case B: $0 \leq q, r$ and, $m \leq q \leq z + m$ or $-(z + m) \leq r \leq -m$.
Compute $(q, r) \leftarrow (q - m, r + z)$, then return (q, r).

Case C: $q, r < 0$ and, $-(z + m) \leq q \leq -m$ or $-(z + m) \leq r \leq -m$.
Compute $(q, r) \leftarrow (q + (n_0 - z), r + (n_1 + m))$, then return (q, r) or go to case B or C.

Case D: $0 \leq q, r$ and, $-(z + m) \leq q \leq -m$ or $m \leq r \leq z + m$.
Compute $(q, r) \leftarrow (q + m, r - z)$, then return (q, r).

In fact, q and r are always in fixed range; $-3 \cdot 2^n < q < 5 \cdot 2^n$ and $-2 \cdot 2^n < r < 2^n$, because of the equation in Algorithm 3; $q = r_3 + r_4 - q_5 - q_6 + q_7$ and $r = -r_5 - r_6 + r_7$ with $-2^n < q_i < 2^n$ and $0 \leq r_i < 2^n$. Therefore, it might happen that q and r are outside the area defined by case A to D. However, we can easily extend our technique to such cases.

7 Conclusion

We proposed a novel technique for $2n$-bit Montgomery multiplications, provided that n-bit Montgomery multiplications are available. We defined the quotient of Montgomery multiplications for $2n$-bit Montgomery multiplications. Since Montgomery multiplications have already been implemented on many platforms, we proposed one technique to emulate the calculation of Montgomery quotients

Table 1. Calculation of a quotient of Montgomery Multiplications

Calls (average)	case 1 : do not change MultMon instruction		case 2 : change MultMon instruction	
modulus	$0 < w < 2^{n-1}$	$2^{n-1} < w < 2^n$	$0 < w < 2^{n-1}$	$2^{n-1} < w < 2^n$
Addition/Subtraction	7.5	5.5	0	2
MultMon instruction	2	2	0	0
MultMonDiv instruction	0	0	1	1

Table 2. Average calls of a $2n$-bit Montgomery Multiplications and an n-bit one

Bitlength	n	$2n$	
Calls(average)	–	case 1	case 2
Addition/Subtraction	0	50.5	12
MultMoninstruction	1	14	0
MultMonDivinstruction	0	0	7

with the available Montgomery multiplications unit. In addition, we proposed another approach where the implementation of the Montgomery multiplications unit is changed in order to directly calculate the quotient. The approach of calling available units takes two instructions, and the approach changing the units has roughly the same cost as one instruction.

As a result, our proposed techniques calculate $2n$-bit Montgomery multiplications by calling the crypto-coprocessor implementing n-bit Montgomery multiplications only, or the instruction for computing remainders and quotients of n-bit Montgomery multiplications.

This paper concentrates on the way to compute double-size Montgomery multiplications with a single-size crypto-coprocessor. Therefore, although the scheme of Chevallier-Mames et al. requires less calls to the multiplier than Fischer et al.'s one, we extended the scheme of Fischer et al., which allows to make our scheme simple: our proposed representation of $2n$-bit integers can avoid using even moduli for Montgomery multipliers. A further direction of this research is to optimize computational costs of $2n$-bit Montgomery multiplications, for example by using Chevallier-Mames et al. technique.

References

1. Chevallier-Mames, B., Joye, M., Paillier, P.: Faster Double-Size Modular Multiplication From Euclidean Multipliers. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 214–227. Springer, Heidelberg (2003)
2. Fischer, W., Seifert, J.P.: Increasing the bitlength of crypto-coprocessors. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 71–81. Springer, Heidelberg (2003)
3. Handschuh, H., Paillier, P.: Smart card crypto-coprocessors for public-key cryptography. In: Schneier, B., Quisquater, J.-J. (eds.) CARDIS 1998. LNCS, vol. 1820, pp. 372–379. Springer, Heidelberg (2000)

4. Koc, C.: Montgomery reduction with even modulus. IEE Proceedings Computer and Digital Techniques 141(5), 314–316 (1994)
5. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
6. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
7. Naccache, D., M'Raïhi, D.: Arithmetic co-processors for public-key cryptography: The state of the art. In: CARDIS, pp. 18–20 (1996)
8. Paillier, P.: Low-cost double-size modular exponentiation or how to stretch your cryptoprocessor. In: Imai, H., Zheng, Y. (eds.) PKC 1999. LNCS, vol. 1560, pp. 223–234. Springer, Heidelberg (1999)
9. Quisquater, J.J., Couvreur, C.: Fast decipherment algorithm for rsa public-key cryptosystem. Electronics Letters 18(21), 905–907 (1982)
10. Rivest, R.L., Shamir, A., Adelman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21(2), 120–126 (1978)
11. RSA Laboratories: RSA challenges, `http://www.rsasecurity.com/rsalabs/`

A Modified Fischer et al.'s Algorithm

We will prove theorem 1 in Section 2.1 to be correct, which an output of Algorithm 1 is indeed congruent to $ABM^{(-1)}$ modulo N, where $0 \leq A, B < N$, $2^{2n-1} < N < 2^{2n}$ and $M = 2^{2n}$.

Proof. Firstly, $2n$-bit integers A, B, N are decomposed on the following equation;

$$A = a_1 z + a_0 m,\ B = b_1 z + b_0 m,\ N = n_1 z + n_0 m$$

where z is odd, $2^{n-1} \leq z < 2^n$ and $m = 2^n$. Then, we continue to be the following.

$$\begin{aligned}
AB &= (a_1 z + a_0 m)(b_1 z + b_0 m) \\
&= a_1 b_1 zz + a_1 b_0 zm + a_0 b_1 zm + a_0 b_0 mm \\
&= a_1 (q_1 n_1 + r_1 m) z + a_1 b_0 zm + a_0 b_1 zm + a_0 b_0 mm \\
&= a_1 r_1 zm - a_1 q_1 n_0 m + a_1 b_0 zm + a_0 b_1 zm + a_0 b_0 mm \\
&= a_1 r_1 zm - a_1 (q_2 z + r_2 m) m + a_1 b_0 zm + a_0 b_1 zm + a_0 b_0 mm \\
&= a_1 (r_1 - q_2 + b_1) zm - a_1 r_2 mm + a_0 b_1 zm + a_0 b_0 mm \\
&= (q_3 n_1 + r_3 m) zm - a_1 r_2 mm + a_0 b_1 zm + a_0 b_0 mm \\
&= (q_3 n_1 + r_3 m) zm - a_1 r_2 mm + (q_4 n_1 z + r_4 m) zm + a_0 b_0 mm \\
&= (r_3 + r_4) zmm - a_1 r_2 mm - (q_3 + q_4) mm + a_0 b_0 mm \\
&= (r_3 + r_4) zmm - a_1 r_2 mm - (q_5 z + r_5 m) mm + a_0 b_0 mm \\
&= (r_3 + r_4) zmm - (q_6 z + r_6 m) mm - (q_5 z + r_5 m) mm + a_0 b_0 mm \\
&= (r_3 + r_4) zmm - (q_6 z + r_6 m) mm - (q_5 z + r_5 m) mm + (q_7 z + r_7 m) mm \\
&= (r_3 + r_4 - q_5 - q_6 + q_7) zmm - (r_5 + r_6 - r_7) mmm \\
&= \{(r_3 + r_4 - q_5 - q_6 + q_7) z - (r_5 + r_6 - r_7) m\} m^2
\end{aligned}$$

□

B Proof of Theorem 2

To show the proof of Theorem 2 telling correctness of Algorithm 4, we prove the following Lemmas 1, 2 and 3, step by step. Lemma 1 states the ranges of the quotient.

Lemma 1. *If* $0 \leq x, y < \min\{w, 2^n\}$, $2^{n-1} < w < 2^{n+1}$, $m = 2^n$, r *is provided by* $\mathsf{MultMon}(x, y, w)$ *and* q *satisfies the equation;* $xy = qw + rm$, *then* $-2^n < q < 2^{n+1}$ *holds.*

Proof. If w is in the range of $2^{n-1} < w < 2^{n+1}$, then we have $qw = xy - rm$ and $-w2^n < xy - rm < 2^{2n}$. Since the minimum value of w is greater than 2^{n-1}, Lemma 1 holds. □

Lemma 2 states ranges of difference between two different quotients of Montgomery multiplications. Whenever Lemma 1 holds, Lemma 2 also holds.

Lemma 2. *If* $0 \leq x, y < w$, $2^{n-1} < w < 2^n$, $m = 2^n$, $gcd(w, m) = 1$, r *is provided by* $\mathsf{MultMon}(x,y,w)$, r' *is provided by* $\mathsf{MultMon}(x,y,w + 2^n)$, q *satisfies the equation;* $xy = qw + rm$ *and* q' *satisfies the equation;* $xy = q'(w + 2^n) + r'm$, *then when we set* δ *as* $\{xy + rw - r'(w + 2^n) \pmod{2^2}\}$, *either* $q' - q = \delta 2^n$ *or* $q' - q = (\delta - 2^2)2^n$ *holds.*

Proof. From Lemma 1, $-2^{n+1} \leq q' - q \leq 2^{n+1}$. Moreover, noticing that $(w + 2^n)^{-1} - w^{-1} = 2^n \pmod{2^{n+2}}$, we get:

$$\begin{aligned} q' - q &= xy((w + 2^n)^{-1} - w^{-1}) - r'm(w + 2^n)^{-1} + rmw^{-1} \\ &\equiv xy2^n - r'(w + 2^n)^{-1}m + rw^{-1}m \pmod{2^{n+2}} \\ &= \{xy - r'(w + 2^n)^{-1} + rw^{-1}\} \times 2^n \end{aligned}$$

Furthermore, $(w + 2^n)^{-1} = w^{-1} = w \pmod{2^2}$, so we have:

$$q' - q \equiv \{xy - r'(w + 2^n) + rw \pmod{2^2}\} \times 2^n \pmod{2^{n+2}}$$

If $(xy - r'(w + 2^n) + rw) < 2 \pmod{2^2}$, then

$$q' - q = \{xy - rw + r'(w + 2^n) \pmod{2^2}\} \times 2^n.$$

Otherwise, we have:

$$q' - q = \{\{xy + rw - r'(w + 2^n) \pmod{2^2}\} - 2^2\} \times 2^n.$$ □

Finally, based on Lemma 2, Lemma 1 shows the equation for getting the quotient of Montgomery multiplications.

Lemma 3. *If* $0 \leq x, y < w$, $2^{n-1} < w < 2^n$, $m = 2^n$, $gcd(w, m) = 1$, r *is provided by* $\mathsf{MultMon}(x,y,w)$, r' *is provided by* $\mathsf{MultMon}(x,y,w + 2^n)$, q *satisfies the equation;* $xy = qw + rm$ *and* q' *satisfies the equation;* $xy = q'(w + 2^n) + r'm$, *then when we set* δ *as* $\{xy + rw - r'(w + 2^n) \pmod{2^2}\}$, *either* $q = \delta(w + 2^n) + r' - r$ *or* $q = (\delta - 2^2)(w + 2^n) + r' - r$ *holds.*

Proof. By hypothesis, $qw+rm = q'(w+2^n)+r'm$. Therefore $q2^n = (q-q')(w+2^n)+(r-r')m$ holds. From lemma 2, we have:

$$\text{either } q2^n = \delta 2^n(w+2^n)+(r'-r)m, \text{ or}$$
$$q2^n = (\delta - 2^2)2^n(w+2^n)+(r'-r)m. \qquad \square$$

The proof of Theorem 2 follows from Lemma 3.

C Approach for Quotients of Montgomery Multiplications Based on limited Memories of a Coprocessor

Lemma 2 and Lemma 3 assume $r' = \mathsf{MultMon}(x, y, w+2^n)$ and q' that satisfy $xy = q'(w+2^n)+r'm$. But it is possible that the MultMon instruction cannot treat the modulus $(w+2^n)$ when it is implemented in hardware with just n-bit memory, or when there are restrictions for the size of the modulus. For this case, we use $(w \pm 2^{n-2})$ rather than $(w+2^n)$. In this case, one can prove lemmas and theorems similar to that from Section 5. We show algorithms to calculate the quotient of Montgomery multiplications instead of Algorithm 4,

Algorithm 6. Implementation of MultMonDiv Limited version1

INPUT: x, y, w with $0 \le x, y < w$, $2^{n-1}+2^{n-2} < w < 2^n$ and $m = 2^n$ with $\gcd(w, m) = 1$;
OUTPUT: q,r;

1. $r \leftarrow \mathsf{MultMon}(x, y, w)$
2. $r' \leftarrow \mathsf{MultMon}(x, y, w-2^{n-2})$
3. $tmp \leftarrow xy + rw - r'(w-2^{n-2}) \pmod{2^4}$
4. if tmp $\ge 2^3$ then $tmp \leftarrow tmp - 2^4$.
5. $q \leftarrow tmp \times (w-2^{n-2}) + 4(r'-r)$
6. **Return** (q, r)

Algorithm 7. Implementation of MultMonDiv Limited version2

INPUT: x, y, w with $0 \le x, y < w$, $2^{n-1} < w < 2^{n-1}+2^{n-2}$ and $m = 2^n$ with $\gcd(w, m) = 1$;
OUTPUT: q,r;

1. $r \leftarrow \mathsf{MultMon}(x, y, w)$
2. $r' \leftarrow \mathsf{MultMon}(x, y, w+2^{n-2})$
3. $tmp \leftarrow xy - rw + r'(w+2^{n-2}) \pmod{2^4}$
4. if tmp $\ge 2^3$ then $tmp \leftarrow tmp - 2^4$.
5. $q \leftarrow tmp \times (w+2^{n-2}) + 4(r-r')$
6. **Return** (q, r)

only when coprocessors have limited memories, whose bit-lengths are just n bits. Our proposed algorithms are divided into Algorithm 6 and Algorithm 7, because we treat $(2^{n-1} < w < 2^{n-1} + 2^{n-2})$ and $(2^{n-1} + 2^{n-2} < w < 2^n)$ separately.

Firstly, we show Algorithm 6, where w is $2^{n-1} < w < 2^{n-1} + 2^{n-2}$.

Secondly, we show Algorithm 7, where w is $2^{n-1} + 2^{n-2} < w < 2^n$.