A Black Hen Lays White Eggs

Bipartite Multiplier Out of Montgomery One for On-Line RSA Verification

Masayuki Yoshino, Katsuyuki Okeya, and Camille Vuillaume

Hitachi, Ltd., Systems Development Laboratory, Kawasaki, Japan
{masayuki.yoshino.aa,katsuyuki.okeya.ue,camille.vuillaume.ch}@hitachi.com

Abstract. This paper proposes novel algorithms for computing double-size modular multiplications with few modulus-dependent precomputations. Low-end devices such as smartcards are usually equipped with hardware Montgomery multipliers. However, due to progresses of mathematical attacks, security institutions such as NIST have steadily demanded longer bit-lengths for public-key cryptography, making the multipliers quickly obsolete. In an attempt to extend the lifespan of such multipliers, double-size techniques compute modular multiplications with twice the bit-length of the multipliers. Techniques are known for extending the bit-length of classical Euclidean multipliers, of Montgomery multipliers and the combination thereof, namely bipartite multipliers. However, unlike classical and bipartite multiplications, Montgomery multiplications involve modulus-dependent precomputations, which amount to a large part of an RSA encryption or signature verification. The proposed double-size technique simulates double-size multiplications based on single-size Montgomery multipliers, and yet precomputations are essentially free: in an 2048-bit RSA encryption or signature verification with public exponent $e = 2^{16} + 1$, the proposal with a 1024-bit Montgomery multiplier is 1.4 times faster than the best previous technique.

Keywords: Montgomery multiplication, double-size technique, RSA, efficient implementation, smartcard.

1 Introduction

The algorithm proposed by Montgomery has been extensively implemented to perform costly modular multiplications which are time-critical for public-key cryptosystems such as RSA [Mon85, RSA78]. In particular, and unlike naive implementations of classical modular multiplications, Montgomery multiplications are not affected by carry propagation delays for computing the quotient of a modular reduction, and as a result, are suitable for high-performance hardware implementations. However, such accelerators are penalized by a strict restriction: their operand size is fixed. In order to deal with recent integer factoring records and ensure long-term security [Len04], official security institutions are updating their standards to longer key sizes than the mainstream 1024 bits for

G. Grimaud and F.-X. Standaert (Eds.): CARDIS 2008, LNCS 5189, pp. 74–88, 2008.

RSA [Nis07, EMV, Ecr06]; unfortunately, such bit lengths are not supported by many cryptographic coprocessors.

This problem has motivated many studies for double-size modular multiplication techniques using single-size hardware modular multipliers. On the one hand, thanks to the Chinese Remainder Theorem, *private* operations (signature generation or decryption) can work with only single-size modular multiplications for computing double-size modular exponentiations [MOV96]. On the other hand, the Chinese Remainder Theorem is no help for *public* operations, and double-size techniques without using private keys are necessary. Following Paillier's seminal paper [Pai99], several solutions were proposed for simulating double-size classical modular multiplications with single-size classical modular multipliers [FS03, CJP03], and later, the techniques were adapted in order to simulate double-size Montgomery multiplications with the commonly used single-size Montgomery multiplier [YOV07a]. Finally, the less common but nonetheless promising bipartite multiplier [KT05], which includes a Montgomery and a classical multiplier working in parallel, was taking advantage of for simulating double-size bipartite multiplications [YOV07b].

In the context of public operations, RSA signature verification for instance, the verifier is unlikely to know the RSA modulus in advance; we refer to this event as *on-line* verification. On the one hand, classical modular multiplications are not affected by the fact that verification is performed off-line or on-line. With a bipartite multiplier, some modulus-dependent precomputations are required during on-line verification. However, when the parameters of the multiplier are appropriately chosen, the cost of precomputations is negligible [KT05]. But on the other hand, precomputations are far from being negligible when using Montgomery multipliers, especially when the public exponent is small. Assuming the 2048-bit exponentiation $X^e \mod Z$, the basis X must be firstly converted to its Montgomery representation, namely $X * 2^{2048} \mod Z$, which can be accomplished with 2048 successive shifts or eleven 2048-bit Montgomery multiplications; in the latter case, this amounts to 36% of the total verification time when $e = 2^{16} + 1$. This is especially unfortunate considering the fact that Montgomery multipliers represent the most popular architecture for cryptographic coprocessors [NM96].

In this paper, we solve the problem of costly on-line precomputations with a radically new approach. Although we assume a multiplier based on the celebrated Montgomery multiplication technique, we simulate a bipartite double-size multiplication, where on-line precomputations are essentially free. Although our double-size bipartite multiplication technique is slightly slower than double-size Montgomery multiplications, the penalty is largely counterbalanced by the benefit in terms of precomputations, at least when the public exponent e is small. When $e = 2^{16} + 1$, which is by far the most common choice for RSA, our technique is 1.4 times faster than the best previous techniques, and even more when $e = 3$. In addition, when the CPU and the coprocessor operate in parallel, which is possible on some low-cost microcontrollers, our proposal can be further optimized, leading to even greater speed. As a consequence, our simulated bipartite

multiplier is the fastest among double-size techniques for cryptographic devices equipped with Montgomery multipliers, and allows the current generation of such multipliers to comply with upcoming key-length standards of official institutes.

Notation: Let ℓ denote operand size of hardware modular multiplication units and L equal to 2ℓ. Small letters such as x, y and z denote ℓ-bit integers, and capital letters such as X, Y and Z denote L-bit integers, where Z is an odd modulus greater than 2^{L-1} like in the case of L-bit RSA.

2 Previous Double-Size Techniques

Montgomery multiplication algorithm has been extensively implemented as cryptographic coprocessors to help low-end devices performing heavy modular multiplications. However, the coprocessors are designed to support the main stream 1024-bit RSA, and face with the upper limit of their bit length to comply with upcoming key-length standards, such as the NIST recommendation; 2048-bit RSA. The problem has motivated double-size techniques to compute modular multiplication with twice the bit length of hardware multipliers.

2.1 Yoshino et al.'s Scheme

This subsection introduces Yoshino et al.'s work[YOV07a, YOV07b]: how to compute a double-size Montgomery multiplication with single-size Montgomery multiplications.

The double-size techniques proposed by Yoshino et al. require not only remainders but also quotients of single-size Montgomery multiplications. The equation $xy = q_m z + r_m c$ shows the relation among products of multiplier x and multiplicand y, quotient q_m and modulus z, and remainder r_m and constant c, where the constant c is usually selected as power of 2 for efficient hardware implementations in practice, therefore this paper also assumes such c satisfying $c = 2^\ell$ [MOV96].

Definition 1 shows an mu instruction for performing single-size Montgomery multiplications, outputting only the remainder.

Definition 1. *For numbers,* $0 \leq x, y < z$ *and* z *is odd, the* mu *instruction is defined as* $r_m \leftarrow \mathsf{mu}(x, y, z)$ *where* $r_m \equiv xyc^{-1} \pmod{z}$.

Their double-size techniques assumed that an mmu instruction is available, which can be emulated with only 2 calls to single-size Montgomery multipliers, and computes the reminder r_m and the quotient q_m of Montgomery multiplications [YOV07a] satisfying the equation $xy = q_m z + r_m c$.

Definition 2. *For numbers,* $0 \leq x, y < z$ *and* z *is odd, the* mmu *instruction is defined as* $(q_m, r_m) \leftarrow \mathsf{mmu}(x, y, z)$ *where* $q_m = (xy - r_m c)/z$ *and* $r_m \equiv xyc^{-1} \pmod{z}$.

Yoshino et al.'s double-size techniques need two steps other than multiplier calls. First, every L-bit integer X, Y and Z is represented with ℓ-bit integers which can be handled by mmu instructions:

$$X = x_1(c-1) + x_0 c,\ Y = y_1(c-1) + y_0 c \text{ and } Z = z_1(c-1) + z_0 c.$$

Second, all quotients q_m and remainders r_m are sequentially gathered from mmu instructions.

Double-size Montgomery multiplications compute a remainder R_m such that $R_m \equiv XYC^{-1} \pmod{Z}$ where $0 \leq X, Y < Z$, and the constant C is called Montgomery constant, and twice bit length of the constant c: $C = 2^L (= c^2)$. Algorithm 1 shows their double-size Montgomery multiplications requiring 6 calls to mmu instructions, and 12 calls to Montgomery multipliers in total.

Algorithm 1. Double-size Montgomery multiplication [YOV07b]

INPUT: X, Y and Z where $0 \leq X, Y < Z$;
OUTPUT: $XYC^{-1} \pmod{Z}$ where $C = 2^L$;

1. $(q_1, r_1) \leftarrow \mathsf{mmu}(x_1, y_1, z_1)$
2. $(q_2, r_2) \leftarrow \mathsf{mmu}(q_1, z_0, c-1)$ //$c = 2^\ell$
3. $(q_3, r_3) \leftarrow \mathsf{mmu}(x_0 + x_1, y_0 + y_1, c-1)$
4. $(q_4, r_4) \leftarrow \mathsf{mmu}(x_0, y_0, c-1)$
5. $(q_5, r_5) \leftarrow \mathsf{mmu}(c-1, -q_2 + q_3 - q_4 + r_1, z_1)$
6. $(q_6, r_6) \leftarrow \mathsf{mmu}(q_5, z_0, c-1)$
7. **return** $(q_2 + q_4 - q_6 - r_1 - r_2 + r_3 - r_4 + r_5)(c-1) + (r_2 + r_4 - r_6)c \pmod{Z}$

Thanks to Algorithm 1, one can set a new MU instruction to compute L-bit Montgomery multiplications such that $R_m \leftarrow \mathsf{MU}(X, Y, Z)$ where $R_m \equiv XYC^{-1} \pmod{Z}$, $0 \leq X, Y < Z$ and $C = 2^L$.

2.2 *L*-Bit RSA Public Operations

The MU instruction (double-size Montgomery multiplications) introduced in last subsection requires twelve single-size multiplications and other basic modular operations; therefore the number of calls to the MU instruction should be as small as possible to get better performance. This subsection explains the contributions and weak points of previous double-size techniques to RSA public operations, which is the most popular application for double-size techniques.

L-bit RSA public operations (signature verification and encryption) employ an L-bit modular exponentiation with a small exponent, following that $X^e \pmod{Z}$, where the ciphertext or signature X, the public modulus Z, and a small public exponent e. The binary method commonly used for RSA public operations computes double-size Montgomery multiplications and squarings according to the bit pattern of the public exponent e. Algorithm 2 shows a left-to-right binary method, which scans e from the most significant bit e_k to the least significant bit e_0 bit-by-bit.

From the view of efficient computation and mathematical security, the exponent used for RSA *public* operations is much smaller than for *private* operations [MOV96, RSA95]. Currently, by far the most common value of the

Algorithm 2. Binary method from the most significant bit

INPUT: X, Z and small public exponent $e = (e_k \cdots e_i \cdots e_0)_2$ where $0 \leq X < Z$;
OUTPUT: $X^e \pmod{Z}$;

1. $Y \leftarrow C^2 \pmod{Z}$ //$C = 2^L$
2. $T \leftarrow \mathsf{MU}(X, Y, Z)$
3. $Y \leftarrow T$
4. **for** i **from** $k-1$ **down to** 0 **do**
   (a) $T \leftarrow \mathsf{MU}(T, T, Z)$ //squaring
   (b) **if** $e_i = 1$, **do**
      i. **if** $i \neq 0$ **then** $T \leftarrow \mathsf{MU}(T, Y, Z)$ //multiplication
      ii. **if** $i = 0$ **then** $T \leftarrow \mathsf{MU}(T, X, Z)$ //multiplication and reduction
5. **return** T

public exponent e is $2^{16} + 1$ having only two 1's in its binary representation ($=(10000000000000001)_2$). In the case of public exponent $e = 2^{16} + 1$, MU instruction is called only 18 times from Step 2 to Step 5 of Algorithm 2. In addition to that, the Algorithm 2 Step 1 seems to be quite cheap, however, this simple modular squaring is seriously expensive for double-size RSA public operations, as it will be explained below.

2.3 Previous Approaches for On-Line Precomputations

There are important differences between *private* and *public* operations: **off-line** precomputations are possible in the former case whereas the latter case requires **on-line** precomputations.

On-line precomputations in Algorithm 2; Step 1 consists of a simple L-bit modular squaring which might look cheap at first sight; however this is not true for low-end devices such as smartcards. There are two known approaches with/without help from Montgomery multipliers; unfortunately, both are seriously slow, and damage performances of double-size techniques on low-end devices.

(1) **Approach with MU Instruction:**
In an attempt to benefit from hardware accelerators, Algorithm 3 employs MU instructions to perform a L-bit modular squaring ($C^2 \pmod{Z}$) using the binary method. Thanks to the cryptographic coprocessor, the approach looks fast, but in fact, the calculation costs are quite heavy: in the case of a 2048-bit modular squaring, Algorithm 3 takes 120 calls to the multiplier, since MU instruction requires 12 calls to the multiplier and is called 10 times by the binary method. As a consequence, the approach with the MU instruction is very costly considering that it only computes a simple modular squaring.

(2) **CPU approach:**
Theoretically, the CPU can compute any-bit modular multiplications without help from hardware accelerators including the L-bit modular squaring

Algorithm 3. L-bit modular squaring ($C^2 \pmod Z$) with MU instructions

INPUT: bitlength Ł $= (L_{L-1} \cdots L_\ell \cdots L_0)_2$ and modulus Z ;
OUTPUT: $C^2 \pmod Z$ where $C = 2^L$;

1. $D \leftarrow 2C \pmod Z$ **and** $T \leftarrow 2C \pmod Z$
2. **for** i **from** $\lfloor \log_2 L \rfloor - 2$ **down to** 0 **do**
   (a) $D \leftarrow \mathsf{MU}(D, D, Z)$
   (b) **if** $L_i = 1$ **then** $D \leftarrow \mathsf{MU}(D, T, Z)$
3. **return** D

($C^2 \pmod Z$). The approach of Algorithm 4 is taken by computers whose CPUs are powerful enough not to need help from hardware accelerators, however, this is not the case for the low-end devices where the performance gap between CPU and arithmetic coprocessor is usually quite large. As a result, Algorithm 4 is practically much slower than Algorithm 3 in these environments.

Algorithm 4. L-bit modular squaring with only CPU instructions

INPUT: bitlength $L = (L_{L-1} \cdots L_\ell \cdots L_0)_2$ and modulus Z;
OUTPUT: $C^2 \pmod Z$ where $C = 2^L$;

1. $D \leftarrow C - Z$
2. **for** i **from** $\ell - 1$ **down to** 0 **do**
   (a) $D \leftarrow 2D$
   (b) **if** $D \geq C$, **then** $D \leftarrow D - Z$.
3. **if** $D \geq Z$, **then** $D \leftarrow D - Z$.
4. **return** D

3 New Double-Size Bipartite Multiplication

L-bit RSA *public* operations require a simple but expensive **on-line** modular-dependent precomputation for low-end devices with ℓ-bit Montgomery multipliers. This section presents new double-size techniques for such environments, which derive their high performance from Montgomery multipliers while eliminating almost all precomputations.

3.1 Overview

The proposal mixes two different modular multiplication algorithms which are executable with the usual Montgomery multipliers. Fig. 1 shows a design of our techniques: first, L-bit integers X, Y and Z are divided into ℓ-bit integers, and inputted to a hardware accelerator outputting the ℓ-bit remainder r_m of Montgomery multiplications. In addition to single-size *Montgomery* multiplications, the new techniques employ single-size *classical* multiplications. Second, their remainders (r_m and r_c) and quotients (q_m and q_c) are computed based on only

the remainder r_m. Last, the remainders and quotients are assembled to build a double-size remainder R satisfying

$$R \equiv XYc^{-1} \pmod{Z},$$

where $0 \leq X, Y < Z$. The new modular multiplication is accompanied by the constant c, which is only half the bit length of the Montgomery constant C, contributing to the fact that our new on-line precomputations can be performed at much cheaper cost.

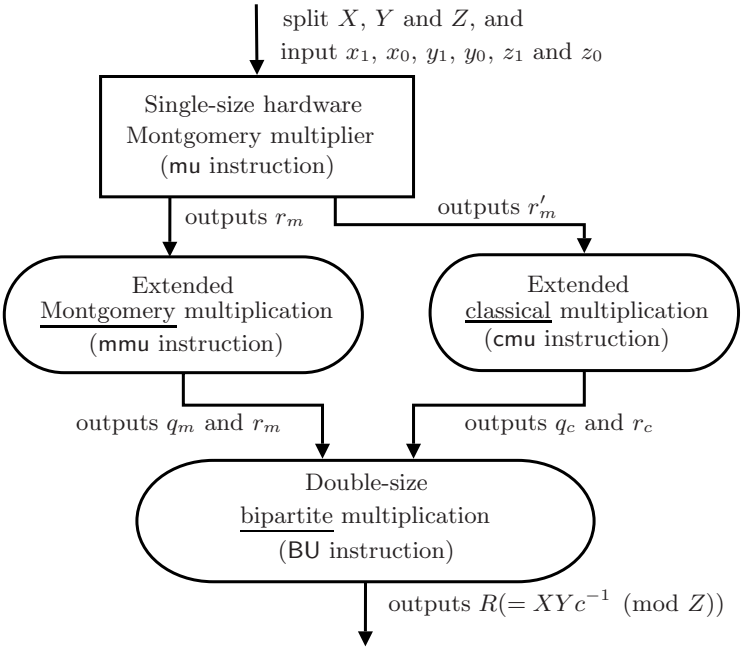

Fig. 1. Configuration of New Double-Size Bipartite Multiplication

3.2 How to Divide L-Bit Integers for the ℓ-Bit Multiplier

In order to benefit from hardware accelerators which can handle only ℓ-bit arithmetic operations, L-bit integers can be simply divided into upper and lower ℓ bits such that $X = x_1c + x_0$, where x_1 is upper and x_0 is lower ℓ bits of X. However, Montgomery multiplications require *odd* moduli[1]. In order to prepare odd moduli, Algorithm 5 derives from the following equation: $Z = z_1c + z_0 = (z_1 + 1)c - (c - z_0)$.

[1] In fact, it is possible to perform Montgomery multiplications with *even* modulus [Koc94]. However, the technique requires other arithmetic operations in addition to the multiplications in hardware: this costly technique is not considered in our paper.

Algorithm 5. L-bit modulus division with odd ℓ-bit moduli

INPUT: odd Modulus Z;
OUTPUT: odd moduli z_1 and z_0 such that $Z = z_1 c + z_0$ with $c = 2^\ell$;

1. $z_1 \leftarrow \lfloor Z/c \rfloor$ **and** $z_0 \leftarrow Z \pmod{c}$.
2. **if** z_1 **is even**, $z_1 \leftarrow z_1 + 1$ **and** $z_0 \leftarrow z_0 - c$.
3. **return** (z_1, z_0)

3.3 New ℓ-Bit Instructions Based on an ℓ-Bit Multiplier

This subsection defines new instructions to output quotients and remainders of classical multiplications and Montgomery multiplications, which can be built on the usual Montgomery multiplier.

Similar with Definition 2 in Section 2.1, the equation; $xy = q_c z + r_c$ shows the relation between the remainder r_c and the quotient q_c of *classical* multiplications, which can be implemented with only three calls to the mu instruction.

Definition 3. *For numbers,* $0 \le x,\ y < z$ *and* z *is odd, the* cmu *instruction is defined as* $(q_c, r_c) \leftarrow \mathsf{cmu}(x, y, z)$ *where* $q_c = (xy - r_c)/z$ *and* $r_c \equiv xy \pmod{z}$.

Algorithm 6 shows how to simulate the cmu instruction with the mu instruction; and the correctness is proven in Appendix A.1.

Algorithm 6. The cmu Instruction based on The mu Instruction

INPUT: x, y, z and t with $0 \le x,\ y < z$, z is odd and $t = c^2 \pmod{z}$;
OUTPUT: q_c and r_c, where $q_c = (xy - r_c)/z$ and $r_c \equiv xy \pmod{z}$;

1. $x' \leftarrow \mathsf{mu}(x, t, z)$ $\quad //\equiv xc \pmod{z}$
2. $r_c \leftarrow \mathsf{mu}(x', y, z)$ $\quad //\equiv xy \pmod{z}$
3. $r'_c \leftarrow \mathsf{mu}(x', y, z + 2)$ $\quad //\equiv xy \pmod{(z+2)}$
4. $q_c \leftarrow (r_c - r'_c)$
5. (a) **if** q_c **is odd**, **then** $q_c \leftarrow (q_c + z + 2)/2$
   (b) **else if** q_c **is even and negative**, **then** $q_c \leftarrow q_c/2 + z + 2$
6. **return** (q_c, r_c)

3.4 How to Build an L-Bit Remainder with ℓ-Bit Instructions

Finally, this subsection presents how to build a remainder of new double-size modular multiplication on the remainders and the quotients of single-size modular multiplications.

Definition 4 shows the BU instruction for computing L-bit *bipartite* multiplication.

Definition 4. *For numbers,* $0 \le X, Y < Z$, *the* BU *instruction is defined as* $R \leftarrow \mathsf{BU}(X, Y, Z)$ *where* $R \equiv XYc^{-1} \pmod{Z}$ *and* $c = 2^\ell$.

The BU instruction performs L-bit modular multiplication; $XYc^{-1} \pmod{Z}$ accompanied with only the ℓ-bit constant c, which is only half the size of the Montgomery constant C, contributing to the fact that our new precomputations can be performed at much cheaper cost.

Algorithm 7 shows how to use the mmu instruction and the cmu instruction to build the BU instruction; the correctness is proven in Appendix A.2.

Algorithm 7. The BU Instruction based on The mmu and cmu Instructions

INPUT: X, Y and Z, where $X = x_1c + x_0, Y = y_1c + y_0$ and $Z = z_1c + z_0$;
OUTPUT: $XYc^{-1} \pmod{Z}$;

1. $(q_1, r_1) \leftarrow \mathsf{cmu}(x_1, y_1, z_1)$
2. $(q_2, r_2) \leftarrow \mathsf{mmu}(q_1, z_0, c-1)$
3. $(q_3, r_3) \leftarrow \mathsf{mmu}(x_0, y_0, c-1)$
4. (a) **if** z_0 **is positive**, $(q_4, r_4) \leftarrow \mathsf{mmu}(q_2 + q_3, c-1, z_0)$
   (b) **if** z_0 **not positive**, $(q_4, r_4) \leftarrow \mathsf{mmu}(-q_2 + q_3, c-1, z_0)$
5. $(q_5, r_5) \leftarrow \mathsf{mmu}(q_4, z_1, c-1)$
6. $(q_6, r_6) \leftarrow \mathsf{mmu}(x_1 + x_0, y_1 + y_0, c-1)$
7. (a) **if** z_0 **is positive**,
   $R \leftarrow (-r_1+r_2+r_3+r_4+q_2+q_3+q_5-q_6)+(r_1-r_2-r_3-r_5+r_6-q_2-q_3-q_5+q_6)c$
   (b) **if** z_0 **is not positive**,
   $R \leftarrow (-r_1-r_2-r_3+r_4-q_2+q_3-q_5-q_6)+(r_1+r_2+r_3+r_5+r_6+q_2-q_3+q_5+q_6)c$
8. **return** $R \pmod{Z}$

Since the cmu instruction based on Algorithm 6 requiring three calls to the Montgomery multiplier is costlier than the mmu instruction requiring only two calls [YOV07a], Algorithm 7 minimizes calls to the cmu instruction, which appears only once in Step 1.

Fig. 2 shows a design of our double-size modular multiplication: the first three lines show components of the product XY, which is computed by the

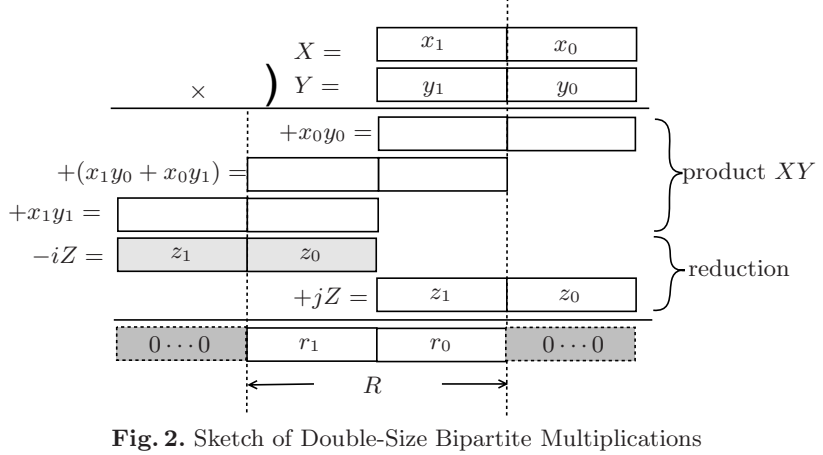

Fig. 2. Sketch of Double-Size Bipartite Multiplications

mmu instruction. There are two kinds of modular reductions in the other steps: One is to subtract Z from the most significant (left) side, which is based on the cmu instruction, and the other adds Z from the least significant (right) side based on the mmu instruction. Finally, one can discard each ℓ-bit integer from the most and least significant side, and get the L-bit remainder located in the middle.

4 Evaluation

This section shows how the proposal speeds up on-line precomputations. NIST recommends using 2048-bit RSA instead of the mainstream 1024-bit RSA from 2010 though 2030 [Nis07]; this paper follows the NIST recommendation, and evaluates the proposed techniques with 2048-bit RSA on smartcards which can only handle 1024-bit Montgomery multiplications.

4.1 Few On-Line Precomputations

L-bit RSA public operations consist of an L-bit modular exponentiation: $X^e \pmod Z$, with the ciphertext or signature X, the public modulus Z, and the small public exponent e. The RSA public operations can be performed by Algorithm 8, which is a left-to-right binary method with the BU instruction presented in Definition 4, and looks similar to Algorithm 2 with the MU instruction requiring heavy precomputations. However, a precomputation of Algorithm 8 (Step 1) is essentially free thanks to the following equation: $c^2 \pmod Z = C - Z$, where $c^2 = C = 2^L$ and $2^{L-1} < Z < 2^L$.

Algorithm 8. Binary method from the most significant bit based on BU instruction

INPUT: X, Z and small public exponent $e = (e_k \cdots e_i \cdots e_0)_2$ where $0 \le X < Z$;
OUTPUT: $X^e \pmod Z$;

1. $Y \leftarrow c^2 \pmod Z$ //$= C - Z$
2. $T \leftarrow \mathsf{BU}(X, Y, Z)$
3. $Y \leftarrow T$
4. **for** i **from** $k-1$ **down to** 0 **do**
   (a) $T \leftarrow \mathsf{BU}(T, T, Z)$ //squaring
   (b) **if** $e_i = 1$, **do**
      i. **if** $i \neq 0$ **then** $T \leftarrow \mathsf{BU}(T, Y, Z)$ //multiplication
      ii. **if** $i = 0$ **then** $T \leftarrow \mathsf{BU}(T, X, Z)$ //multiplication and reduction
5. **return** T

The BU instruction requires other on-line precomputation $c^2 \pmod z$ for cmu instruction, which is called at Algorithm 7 Step 1. This precomputation can easily be performed using Algorithm 9 with only several calls to the hardware multiplier.

Algorithm 9. ℓ-bit modular squaring with mu instructions

INPUT: bitlength $\ell = (\ell_{\ell-1} \cdots \ell_i \cdots \ell_0)_2$ and modulus z ;
OUTPUT: $c^2 \pmod z$ where $c = 2^\ell$;

1. $d \leftarrow 2c \pmod z$ **and** $t \leftarrow 2c \pmod z$
2. **for** i **from** $\lfloor \log_2 \ell \rfloor - 2$ **down to** 0 **do**
   (a) $d \leftarrow \mathsf{mu}(d, d, z)$
   (b) **if** $\ell_i = 1$ **then** $d \leftarrow \mathsf{mu}(d, t, z)$
3. **return** d

4.2 Performance Improvement

The proposed double-size techniques are evaluated for smartcards which can only handle 1024-bit Montgomery multiplications in the case of 2048-bit RSA public operations with the common exponent $e = 2^{16} + 1$ to follow the NIST recommendation [Nis07]. Table 1 includes the performance of the 2048-bit RSA with three columns; on-line precomputations, a modular exponentiation and the total, which are evaluated by the binary (square-and-multiply) methods following Algorithm 2 or Algorithm 8.

The proposal eliminates almost all on-line precomputations, and contributes to improve the total performance: One of the on-line precomputation; $C \pmod Z$ is replaced with a subtraction; $C - Z$, and the other precomputation; $c^2 \pmod z$ requires only 9 calls to the Montgomery multipliers, therefore the proposal advantages in the on-line precomputations. As a result, the proposed method costs only 70%($\simeq 243/336$) of the best previous method.

Figure 3 depicts how the cost, expressed in number of calls to the single-size Montgomery multiplier, varies with the exponent e in the case of a 2048-bit RSA encryption. For exponents of less than 32 bits, our proposal is always better than previous techniques. The turnover when double-size Montgomery multiplications [YOV07a] becomes more competitive than our proposal occurs for the 65-bit exponent $e = (11 \ldots 11)_2$. However, we argue that in practice, small RSA exponents of less than 32 bits represent the overwhelming majority of cases [RSA95].

4.3 Further Performance Improvement

Some micro processor can perform modular operations in parallel with help of cryptographic coprocessors and CPU: while the coprocessors work, CPU can compute other arithmetic modular operations. Despite gap between the speed

Table 1. Calls to the multiplier in the 2048-bit RSA public operation

Scheme	On-line precomputations	Modular exponentiation	Total
[YOV07a]	140	252	392
[YOV07b]	120	216	336
This paper	9	234	243

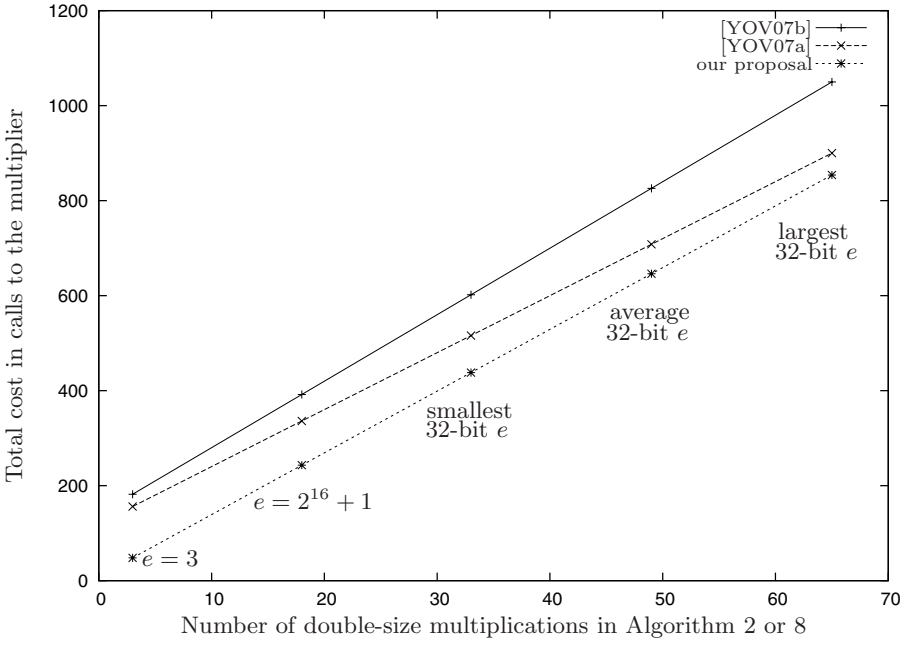

Fig. 3. Calls to the Montgomery multiplier for several exponents e

of those processors, such environments can accelerate double-size modular multiplication assigning some step of Algorithm 8 in the arithmetic processor and the other steps in CPU such as Step 1–5 in the coprocessor and Step 6 in CPU, or Step 1,2,4,5 in the coprocessor and the others in CPU. Therefore, parallel operations help to optimize our proposal, leading to even greater speed.

5 Conclusion

This paper proposed novel double-size modular multiplication algorithms with few modulus-dependent precomputations for on-line RSA public operations, which gave birth to double-size bipartite multiplication on the most commonly used single-size Montgomery multipliers in order to eliminate heavy precomputations required by all previous double-size Montgomery multiplication techniques. Although the proposed double-size bipartite multiplication technique is slightly slower than the best technique of double-size Montgomery multiplication, the penalty is largely counterbalanced by the benefit in terms of precomputations: when the public exponent is $e = 2^{16} + 1$, which is by far the most common choice for RSA, our method is 1.4 times faster than the best previous techniques. In addition, when the CPU and the coprocessor operate in parallel, which is possible for some low-cost micro controllers, our proposal can be further optimized, leading to even greater speed. As a consequence, our double-size bipartite multiplication technique is the fastest among all double-size techniques for the cryptographic devices equipped with hardware Montgomery multipliers.

References

[Koc94] Koç., Ç.K.: Montgomery Reduction with Even Modulus. IEE Proceedings - Computers and Digital Techniques 141(5), 314–316 (1994)

[CJP03] 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)

[Ecr06] European Network of Excellence in Cryptology (ECRYPT). ECRYPT Yearly Report on Algorithms and Keysizes (2006), http://www.ecrypt.eu.org/documents/D.SPA.21-1.1.pdf

[EMV] EMV. EMV Issuer and Application Security Guidelines, Version 2.1 (2007), http://www.emvco.com/specifications.asp?show=4

[FS03] 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)

[KT05] Kaihara, M.E., Takagi, N.: Bipartite modular multiplication. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 201–210. Springer, Heidelberg (2005)

[Len04] Arjen, K.: Lenstra. Key Lengths (2004), http://cm.bell-labs.com/who/akl/key_lengths.pdf

[Mon85] Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)

[MOV96] Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)

[Nis07] National Institute of Standards ant Technology. NIST Special Publication 800-57 Recommendation for KeyManagement Part 1: General (Revised) (2007), http://csrc.nist.gov/CryptoToolkit/tkkeymgmt.html

[NM96] Naccache, D., M'Raïhi, D.: Arithmetic co-processors for public-key cryptography: The state of the art. In: CARDIS, pp. 18–20 (1996)

[Pai99] 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)

[RSA78] 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)

[RSA95] RSA Laboratories. The Secure Use of RSA. CryptoBytes 1(3) (1995), ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto1n3.pdf

[YOV07a] Yoshino, M., Okeya, K., Vuillaume, C.: Unbridle the Bit-Length of a Crypto-Coprocessor with Montgomery Multiplication. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 188–202. Springer, Heidelberg (2007)

[YOV07b] Yoshino, M., Okeya, K., Vuillaume, C.: Double-Size Bipartite Modular Multiplication. In: Pieprzyk, J., Ghodosi, H., Dawson, E. (eds.) ACISP 2007. LNCS, vol. 4586, pp. 230–244. Springer, Heidelberg (2007)

A Proof for Correctness

A.1 Algorithm 6: The cmu Instruction Based on a mu Instruction

Algorithm of Montgomery multiplications is different from classical multiplications; however, one can simulate classical multiplications easily using the ℓ-bit mu instruction implementing Montgomery multiplications thanks to the following equation:

$$r_c = xy \pmod z = x'yc^{-1} \pmod z$$

where $0 \leq x, y < z$ and $x' = xc \pmod z$. Therefore, the mu instruction can output the classical remainder r_c according to the following two intuitive steps:

1. $x' \leftarrow \mathsf{mu}(x, c^2 \pmod z)$[i]
2. $r_c \leftarrow \mathsf{mu}(x', y, z)$

Thanks to these steps, one can compute r_c with help from the multipliers.

There is a requirement for Montgomery multiplications: only odd moduli are available. The following proof show how to compute classical quotient $q_c = (xy - r_c)/z$ from two different classical remainders.

Proof. For numbers, where $0 \leq x$, $y < z$ and z is odd, classical multiplication outputs a quotient q_c and a remainder r_c, which satisfy the following equation: $xy = q_c z + r_c$ where $q_c = (xy - r_c)/z$ and $r_c \equiv xy \pmod z$. Equivalently,

$$\begin{aligned} xy &= q_c z + r_c \\ &= q_c(z+2) + (-2q_c + r_c) \quad (1) \\ &= q'_c(z+2) + r'_c \quad (2) \end{aligned}$$

From the equation (1) and (2),

$$q_c = (r_c - r'_c + \delta(z+2))/2$$

holds with some integer δ.

Since $0 \leq x$,$y < z$ holds, q_c, r_c and r'_c satisfy the following conditions: $0 \leq q_c < z$, $0 \leq r_c < z$ and $0 \leq r'_c < z+2$. From the equation $-(z+2) < (r_c - r'_c) < z$, the following condition holds:

$$\text{If value of } (r_c - r'_c) \text{ is } \begin{cases} \text{even and non negative, then} & \delta = 0 \\ \text{odd, then} & \delta = 1 \\ \text{even and negative, then} & \delta = 2 \end{cases}$$ □

A.2 Algorithm 7: The BU Instruction Based on a mmu and cmu Instruction

Algorithm 7 builds the BU instruction on a cmu instruction and an mmu instruction, and needs to process branches in Step 4 and Step 7 whether z_0 is positive or not. This paper only introduces a proof in the case that z_0 is positive, but one can follow the other case similarly.

[i] Algorithm 9 can help to precompute the equation $c^2 \pmod z$.

Proof. L-bit modulus Z is represented by Algorithm 5 as the followings:

$$Z = z_1 c + z_0$$

where $0 < z_1 < c$ and $-c < z_0 < c$, and the other L-bit integers X and Y are simply divided into upper and lower ℓ-bit integers.

$$X = x_1 c + x_0 \text{ and } Y = y_1 c + y_0.$$

where $0 \leq x_1, x_0, y_1, y_0 < c$. Then, the following equation holds.

$$XY = x_1 y_1 c(c-1) + (x_1 + x_0)(y_1 + y_0)c - x_0 y_0 (c-1) \tag{3}$$

The first term of Equation (3) is transformed into the following equations with the first call to a cmu instruction and the second call to an mmu instruction.

$$\begin{aligned} x_1 y_1 c(c-1) &= (q_1 z_1 + r_1)c(c-1) \\ &\equiv (-q_1 z_0 + r_1 c)(c-1) \quad (\because z_1 c \equiv -z_0 \pmod{Z}) \\ &= (q_2 + (r_1 - r_2 - q_2)c)(c-1) \end{aligned}$$

The third term of Equation (3) is also transformed into the following equation with a call to the mu instruction.

$$x_0 y_0 (c-1) = (-q_3 + (r_3 + q_3)c)(c-1)$$

Therefore, the first and third term of Equation (3) are combined with twice the help from the mmu instruction.

$$\begin{aligned} & x_1 y_1 c(c-1) + x_0 y_0 (c-1) \\ &\quad = (q_2 + q_3)(c-1) + (r_1 - r_2 - r_3 - q_2 - q_3)c(c-1) \\ &\quad = (q_4 z_0 + r_4 c) + (r_1 - r_2 - r_3 - q_2 - q_3)c(c-1) \\ &\quad \equiv (-q_4 z_1 + r_4)c + (r_1 - r_2 - r_3 - q_2 - q_3)c(c-1) \quad (\because z_0 \equiv -z_1 c \pmod{Z}) \\ &\quad = ((q_5 + r_4) - (q_5 + r_5)c)c + (r_1 - r_2 - r_3 - q_2 - q_3)c(c-1) \end{aligned}$$

The second term of Equation (3) is transformed into the followings with last call to the mu instruction.

$$(x_1 + x_0)(y_1 + y_0)c = (-q_6 + (r_6 + q_6)c)c$$

Finally, Equation (3) consisting of three terms are concluded with the following equations.

$$\begin{aligned} XY \equiv \quad & (-r_1 + r_2 + r_3 + r_4 + q_2 + q_3 + q_5 - q_6) \\ & + (r_1 - r_2 - r_3 - r_5 + r_6 - q_2 - q_3 - q_5 + q_6)c \pmod{Z}. \end{aligned}$$

□