# Faster Double-Size Modular Multiplication From Euclidean Multipliers

[Published in C.D. Walter, Ç.K. Koç, and C. Paar, Eds., *Cryptographic Hardware and Embedded Systems – CHES 2003*, vol. 2779 of *Lecture Notes in Computer Science*, pp. 214–227, Springer-Verlag, 2003.]

Benoît Chevallier-Mames, Marc Joye, and Pascal Paillier

[1] Gemplus, Card Security Group
La Vigie, Avenue du Jujubier, ZI Athélia IV, 13705 La Ciotat Cedex, France
{benoit.chevallier-mames, marc.joye}@gemplus.com
[2] Gemplus, Cryptography Group
34 rue Guynemer, 92447 Issy-les-Moulineaux, France
pascal.paillier@gemplus.com

http://www.gemplus.com/smart/

**Abstract.** A novel technique for computing a $2n$-bit modular multiplication using $n$-bit arithmetic was introduced at CHES 2002 by Fischer and Seifert. Their technique makes use of an Euclidean division based instruction returning not only the remainder but also the integer quotient resulting from a modular multiplication, *i.e.*, on input $x, y$ and $z$, both $\lfloor xy/z \rfloor$ and $xy \bmod z$ are returned. A second algorithm making use of a special modular 'multiply-and-accumulate' instruction was also proposed.

In this paper, we improve on these algorithms and propose more advanced computational strategies with fewer calls to these basic operations, bringing in a speed-up factor up to 57%. Besides, when Euclidean multiplications themselves have to be emulated in software, we propose a specific modular multiplication based algorithm which surpasses original algorithms in performance by 71%.

**Keywords.** Modular multiplication, crypto-processors, embedded cryptographic software, efficient implementations, RSA.

## 1 Introduction

When a cryptographic coprocessor is inherently limited to handle numbers of a specific bitsize $n$, performing modular arithmetic operations over larger operands turns out to be an intricate implementation problem. One may think of natural and simple solutions like programming multi-precision algorithms such as those of Montgomery [5], Barrett [3], Quisquater [8] or Walter [10]. These algorithms as well as others, however, require processing data blocks via smaller operations that may not be supported by the underlying hardware architecture. A typical example lies in the regular $(n \times n)$-bit integer multiplication (with a $2n$-bit result), which may not be directly available on a crypto-processor like Infineon's ACE where only $n$-bit modular operations —for $n$ up to 1100— are programmable. To remedy this, a conventional trick tells us to take a blocksize $n \leq 1100/2$ (say $n = 512$), so that integer multiplications result from multiplications modulo $2^{1100}$. Adopting this strategy, a Montgomery-based implementation for a single 2048-bit multiplication would cost no less than forty 512-bit integer multiplications, an unacceptable performance. So implementing a 2048-bit RSA while sustaining higher expectations in terms of execution speed is not a straightforward task on this platform. Given that context, one has to devise more specific techniques to emulate modular arithmetic operations over operands of larger sizes.

Surprisingly enough, little is known about software strategies that would overcome this hardware-originated length limitation in a very efficient way. Two different techniques, however, have appeared in the literature recently. In [6], Paillier presents a $2n$-bit modular multiplication emulated with $8 \to 6$ calls to a modular multiplier of bitsize $n$ (plus other, negligible operations). Paillier's algorithm, inspired by Montgomery's technique [5], strongly relies on Residue Number Systems (RNS) [7, 9] for representing data and performing partial operations on them. It simplifies earlier, more intricate approaches making use of mixed base representations [1, 2]. The efficiency of this system is due to the use of fast base extensions in connection with a specific choice for the RNS base, a choice which also ensures that the result of the double-size multiplication is returned under a representation compatible with the input operands themselves, thereby allowing repeated invocations of the algorithm. Unfortunately, its Montgomery-like style forces one to precompute a modulus-dependent constant prior to multiplying any data.

More recently, in [4], Fischer and Seifert suppress the need for precomputed constants: $2n$-bit operands are handled through a classical radix representation with base $2^n$, the new technique outputting a result under the same representation. Independently, in this work, Fischer and Seifert replace the basic operation with an Euclidean multiplication, *i.e.*, an operation that simultaneously returns both the quotient and remainder of the division $xy \div z$, given arbitrary $n$-bit integers $x, y$ and $z$. The motivation for this stems from the ease of integrating such an operation in a hardware architecture which already supports modular multiplications. On most architectures indeed, the arithmetic units involved in the execution of a modular reduction could be easily enriched to simultaneously output quotient bits with extremely moderate extra cost.

In this paper, we improve on Fischer and Seifert's algorithms and propose more advanced computational strategies with fewer calls to the Euclidean multiplication. Our improved algorithms use $2^n$-radix representations of numbers in the spirit of [4]. We also show that adapting the choice of the radix base according to the modulus may further speed up our technique. This modification can be carried out while maintaining inputs and outputs under the same arithmetic format, thereby making it possible to iterate executions. In the most favorable case, we emulate a double-size modular multiplication with no more than 3 Euclidean multiplications, which leads to a speedup factor of $\frac{7-3}{7} \approx 57\%$. In addition, when Euclidean multiplications themselves must be emulated in software from modular multiplications, we show how to use these directly without referring to RNS-based approaches $[1, 2, 7, 9]$. More precisely, we propose a simple alternative to these works which keeps numbers under a radix representation and runs as fast as 2 Euclidean multiplications. This accelerates original algorithms by $\frac{14-4}{14} \approx 71\%$. We remind that, whatever the computational strategy, it is agreed that $n$-bit linear operations such as signed additions, subtractions, xors, conditional branchings and so forth, are always available and that their respective running times remain negligible in comparison with operations of multiplicative nature. As usual, we consider these as being virtually free operations throughout the paper.

The rest of this paper is organized as follows. In the next section, we review the technique introduced by Fischer and Seifert for emulating a $2n$-bit modular multiplication and show how to improve it in Section 3. Then, in Section 4, we investigate the influence of data representations on the performances of our algorithms. In Section 5, we detail an implementation of an Euclidean multiplication. Section 6 describes a specific strategy for cases when Euclidean multiplications are emulated in software. Finally, we summarize and compare our results in Section 7.

## 2 Fischer and Seifert's Algorithms

Fischer and Seifert's technique [4] relies on the two basic instructions

$$\texttt{MultModDiv}(x, y, z) \stackrel{\Delta}{=} (\lfloor (x \cdot y)/z \rfloor, (x \cdot y) \bmod z) \tag{1}$$

and

$$\texttt{MultModDivInit}(x, y, t, z) \stackrel{\Delta}{=} (\lfloor (x \cdot y + t \cdot 2^n)/z \rfloor, (x \cdot y + t \cdot 2^n) \bmod z)\,, \tag{2}$$

where $x, y, t, z$ are $n$-bit integers. It is implicitly required in [4] that operands $x, y$ and $t$ can be negative, *i.e.*, that the processor is able to handle them whatever their sign through a signed representation without affecting computation results. In fact, these two instructions should, by extension, work for any non-reduced inputs $x, y, t$, namely whenever $|x| > z$ for instance, provided that $\lceil |x|/z \rceil$ remains an extremely small value. Subsequent hardware or software corrections are neglected in the description of all algorithms, as proposed in [4].

The algorithms originally proposed by Fischer and Seifert, which we denote by FS1 and FS2, are depicted on Fig. 1 and Fig. 2, respectively. We refer the reader to [4] for proofs of correctness.

**Input:** $2n$-bit integers $A = A_1 2^n + A_0, B = B_1 2^n + B_0, N = N_1 2^n + N_0$
**Output:** $AB \pmod N$
**Cost:** 7 `MultModDiv`

$(Q^{(1)}, R^{(1)}) = \texttt{MultModDiv}(B_1, 2^n, N_1)$
$(Q^{(2)}, R^{(2)}) = \texttt{MultModDiv}(Q^{(1)}, N_0, 2^n)$
$(Q^{(3)}, R^{(3)}) = \texttt{MultModDiv}(A_1, R^{(1)} - Q^{(2)} + B_0, N_1)$
$(Q^{(4)}, R^{(4)}) = \texttt{MultModDiv}(A_0, B_1, N_1)$
$(Q^{(5)}, R^{(5)}) = \texttt{MultModDiv}(Q^{(3)} + Q^{(4)}, N_0, 2^n)$
$(Q^{(6)}, R^{(6)}) = \texttt{MultModDiv}(A_1, R^{(2)}, 2^n)$
$(Q^{(7)}, R^{(7)}) = \texttt{MultModDiv}(A_0, B_0, 2^n)$
Return $(R^{(3)} + R^{(4)} - Q^{(5)} - Q^{(6)} + Q^{(7)})2^n + (R^{(7)} - R^{(6)} - R^{(5)})$

**Fig. 1.** Fischer-Seifert's modular multiplication algorithm FS1

**Input:** $2n$-bit integers $A = A_1 2^n + A_0, B = B_1 2^n + B_0, N = N_1 2^n + N_0$
**Output:** $AB \pmod N$
**Cost:** 5 `MultModDiv` + 1 `MultModDivInit`

$(Q^{(1)}, R^{(1)}) = \texttt{MultModDiv}(A_1, B_1, N_1)$
$(Q^{(2)}, R^{(2)}) = \texttt{MultModDivInit}(N_0, -Q^{(1)}, R^{(1)}, N_1)$
$(Q^{(3)}, R^{(3)}) = \texttt{MultModDiv}(A_1, B_0, N_1)$
$(Q^{(4)}, R^{(4)}) = \texttt{MultModDiv}(A_0, B_1, N_1)$
$(Q^{(5)}, R^{(5)}) = \texttt{MultModDiv}(A_0, B_0, 2^n)$
$(Q^{(6)}, R^{(6)}) = \texttt{MultModDiv}(Q_{(2)} + Q^{(3)} + Q^{(4)}, N_0, 2^n)$
Return $(R^{(2)} + R^{(3)} + R^{(4)} + Q^{(5)} - Q^{(6)})2^n + (R^{(5)} - R^{(6)})$

**Fig. 2.** Fischer-Seifert's modular multiplication algorithm FS2

## 3 Improved Algorithms

Our idea consists in rewriting the modular multiplication in terms of manipulations over half-size operands. This is reminiscent of Karatsuba's famous method which we recall here for the sake of completeness.

**Lemma 1 (Karatsuba).** *If $A = A_1 2^n + A_0$ and $B = B_1 2^n + B_0$ then*

$$AB = 2^n(2^n - 1)A_1B_1 + 2^n(A_1 + A_0)(B_1 + B_0) - (2^n - 1)A_0B_0 \enspace .$$

Our first algorithm only makes use of Fischer and Seifert's `MultModDiv` instruction while our second algorithm also employs the `MultModDivInit` instruction.

### 3.1 Using `MultModDiv` Instructions Only

In this section, we eliminate a `MultModDiv` instruction in Fischer-Seifert's technique. We state:

**Theorem 1.** *Given arbitrary $2n$-bit integers $N$ and $A, B \leq N$, the $2n$-bit integer $AB \bmod N$ can be computed with at most <u>six</u> $n$-bit `MultModDiv` instructions.*

Our algorithm, denoted A1, is described hereafter on Fig. 3.

| | |
|---|---|
| **Input:** | $2n$-bit integers $A = A_1 2^n + A_0, B = B_1 2^n + B_0, N = N_1 2^n + N_0$ |
| **Output:** | $AB \pmod N$ |
| **Cost:** | 6 `MultModDiv` |

$(Q^{(1)}, R^{(1)}) = \mathtt{MultModDiv}(A_1, B_1, N_1)$
$(Q^{(2)}, R^{(2)}) = \mathtt{MultModDiv}(Q^{(1)}, N_0, 2^n)$
$(Q^{(3)}, R^{(3)}) = \mathtt{MultModDiv}(A_1 + A_0, B_1 + B_0, 2^n - 1)$
$(Q^{(4)}, R^{(4)}) = \mathtt{MultModDiv}(A_0, B_0, 2^n)$
$(Q^{(5)}, R^{(5)}) = \mathtt{MultModDiv}(2^n - 1, R^{(1)} + Q^{(3)} - Q^{(2)} - Q^{(4)}, N_1)$
$(Q^{(6)}, R^{(6)}) = \mathtt{MultModDiv}(Q^{(5)}, N_0, 2^n)$
Return $(R^{(3)} + R^{(5)} - Q^{(6)} - R^{(2)} - R^{(4)})2^n + (R^{(2)} + R^{(4)} - R^{(6)})$

**Fig. 3.** Our improved algorithm A1 for double-size modular multiplication

*Proof (of correctness for* A1*).* For convenience, we write $Z = 2^n$ and denote by $\equiv_N$ the equivalences modulo $N$. Then, rewriting Lemma 1 gives

$$AB = Z(Z - 1)A_1B_1 + Z(A_1 + A_0)(B_1 + B_0) - (Z - 1)A_0B_0 \enspace .$$

Moreover, noticing that $N_1 Z \equiv_N -N_0$, we get

$$\begin{aligned} Z(Z-1)A_1B_1 &\equiv_N Z(Z-1)(Q^{(1)}N_1 + R^{(1)}) \\ &\equiv_N -(Z-1)(Q^{(1)}N_0) + Z(Z-1)R^{(1)} \\ &\equiv_N -(Z-1)(Q^{(2)}Z + R^{(2)}) + Z(Z-1)R^{(1)} \\ &\equiv_N Z(Z-1)(R^{(1)} - Q^{(2)}) - (Z-1)R^{(2)}\,, \end{aligned}$$

$$Z(A_1 + A_0)(B_1 + B_0) = Z((Z-1)Q^{(3)} + R^{(3)}) = Z(Z-1)Q^{(3)} + ZR^{(3)}$$

and

$$(Z-1)A_0B_0 = (Z-1)(ZQ^{(4)} + R^{(4)}) = Z(Z-1)Q^{(4)} + (Z-1)R^{(4)} \ .$$

Hence, we have

$$\begin{aligned} AB &\equiv_N Z(Z-1)(R^{(1)} + Q^{(3)} - Q^{(2)} - Q^{(4)}) + ZR^{(3)} - (Z-1)(R^{(2)} + R^{(4)}) \\ &\equiv_N Z(Q^{(5)}N_1 + R^{(5)}) + ZR^{(3)} - (Z-1)(R^{(2)} + R^{(4)}) \\ &\equiv_N -Q^{(5)}N_0 + Z(R^{(3)} + R^{(5)}) - (Z-1)(R^{(2)} + R^{(4)}) \\ &\equiv_N -(Q^{(6)}Z + R^{(6)}) + Z(R^{(3)} + R^{(5)}) - (Z-1)(R^{(2)} + R^{(4)}) \\ &\equiv_N (R^{(3)} + R^{(5)} - Q^{(6)} - R^{(2)} - R^{(4)})Z + (R^{(2)} + R^{(4)} - R^{(6)})\,, \end{aligned}$$

which proves the correctness of Algorithm A1 and of Theorem 1. □

### 3.2 Using `MultModDiv` and `MultModDivInit` Instructions

Here again, we invoke Lemma 1 and improve Fischer and Seifert's double size multiplier FS2. Formally, we state:

**Theorem 2.** *Given arbitrary $2n$-bit integers $N$ and $A, B \le N$, the $2n$-bit integer $AB \bmod N$ can be computed with at most four $n$-bit* `MultModDiv` *instructions and one $n$-bit* `MultModDivInit` *instruction.*

Our algorithm, denoted A2, is described below on Fig. 4.

| | |
|---|---|
| **Input:** | $2n$-bit integers $A = A_1 2^n + A_0, B = B_1 2^n + B_0, N = N_1 2^n + N_0$ |
| **Output:** | $AB \pmod N$ |
| **Cost:** | 4 `MultModDiv` + 1 `MultModDivInit` |

$(Q^{(1)}, R^{(1)}) = \texttt{MultModDiv}(A_1, B_1, N_1)$
$(Q^{(2)}, R^{(2)}) = \texttt{MultModDiv}(A_1 + A_0, B_1 + B_0, 2^n - 1)$
$(Q^{(3)}, R^{(3)}) = \texttt{MultModDiv}(A_0, B_0, 2^n)$
$(Q^{(4)}, R^{(4)}) = \texttt{MultModDivInit}(Q^{(1)}, N_0, Q^{(3)} - R^{(1)} - Q^{(2)}, N_1)$
$(Q^{(5)}, R^{(5)}) = \texttt{MultModDiv}(N_0 + N_1, Q^{(4)}, 2^n)$
Return $(R^{(2)} + Q^{(5)} - R^{(3)} - R^{(4)})2^n + (R^{(3)} + R^{(4)} + R^{(5)})$

**Fig. 4.** Improved algorithm A2 for double-size modular multiplication

*Proof (of correctness for* A2*).* As before, we set $Z = 2^n$. We have

$$
\begin{aligned}
Z(Z-1)A_1B_1 &\equiv_N Z(Z-1)(Q^{(1)}N_1 + R^{(1)}) \equiv_N (Z-1)(-Q^{(1)}N_0 + R^{(1)}Z)\,,\\
Z(A_1+A_0)(B_1+B_0) &= Z((Z-1)Q^{(2)} + R^{(2)}) = (Z-1)Q^{(2)}Z + ZR^{(2)}\,,\\
(Z-1)A_0B_0 &= (Z-1)(ZQ^{(3)} + R^{(3)}) = (Z-1)Q^{(3)}Z + (Z-1)R^{(3)}
\end{aligned}
$$

so that

$$
\begin{aligned}
AB &\equiv_N -(Z-1)\big(Q^{(1)}N_0 + (Q^{(3)} - R^{(1)} - Q^{(2)})Z\big) + ZR^{(2)} - (Z-1)R^{(3)}\\
&\equiv_N -(Z-1)(Q^{(4)}N_1 + R^{(4)}) + ZR^{(2)} - (Z-1)R^{(3)}\\
&\equiv_N (N_0+N_1)Q^{(4)} + ZR^{(2)} - (Z-1)(R^{(3)} + R^{(4)})\\
&\equiv_N (Q^{(5)}Z + R^{(5)}) + ZR^{(2)} - (Z-1)(R^{(3)} + R^{(4)})\\
&\equiv_N (R^{(2)} + Q^{(5)} - R^{(3)} - R^{(4)})Z + (R^{(3)} + R^{(4)} + R^{(5)})
\end{aligned}
$$

since $N_1(Z-1) \equiv_N -N_0 - N_1$. □

## 4 Further Improvements Using Specific Representations

All algorithms considered so far manipulate integers in radix representation with base $2^n$. We now show how changing that representation may lead to further cost savings in our algorithms. Although we explicitly describe only a couple of (modulus-dependent) representations in what follows, there might exist other ones which would reveal quite as efficient. In both cases, the idea is simply to employ a clever representation base derived from modulus $N$. This computation is performed prior to the execution of the corresponding double-size modular multiplication and can be executed once and for all, especially when the multiplication is invoked repeatedly.

### 4.1 Down to 5 `MultModDiv`

Let $X = \lceil\sqrt{N}\rceil$. Then setting $\alpha = X^2 \bmod N$, we have $\alpha < 2X$. We state:

**Theorem 3.** *Given arbitrary $2n$-bit integers $N$ and $A, B \le N$, the $2n$-bit integer $AB \bmod N$ can be computed with at most five $n$-bit* `MultModDiv` *instructions.*

We denote our new algorithm by A3 and describe it on Fig. 5.

*Proof (of Algorithm* A3*).* Using $X^2 \equiv_N \alpha$ and

$$AB = X(X-1)A_1B_1 + X(A_1+A_0)(B_1+B_0) - (X-1)A_0B_0\ \ ,$$

we get

$$
\begin{aligned}
(X-1)A_0B_0 &\equiv_N (X-1)(Q^{(1)}X + R^{(1)}) \equiv_N Q^{(1)}\alpha - R^{(1)} + (R^{(1)} - Q^{(1)})X\,,\\
X(A_1+A_0)(B_1+B_0) &\equiv_N X(Q^{(2)}X + R^{(2)}) \equiv_N Q^{(2)}\alpha + R^{(2)}X\,,
\end{aligned}
$$

| | |
|---|---|
| **Input:** | radix base $X$, $2n$-bit integers $A = A_1X + A_0, B = B_1X + B_0$ |
| **Output:** | $AB \pmod{N}$ |
| **Cost:** | 5 `MultModDiv` |

$(Q^{(1)}, R^{(1)}) = \texttt{MultModDiv}(A_0, B_0, X)$
$(Q^{(2)}, R^{(2)}) = \texttt{MultModDiv}(A_1 + A_0, B_1 + B_0, X)$
$(Q^{(3)}, R^{(3)}) = \texttt{MultModDiv}(A_1, B_1, X)$
$(Q^{(4)}, R^{(4)}) = \texttt{MultModDiv}(\alpha, Q^{(3)}, X)$
$(Q^{(5)}, R^{(5)}) = \texttt{MultModDiv}(\alpha, -Q^{(1)} + Q^{(2)} - Q^{(3)} + Q^{(4)} + R^{(3)}, X)$
Return $(R^{(5)} + R^{(1)}) + (R^{(4)} - R^{(1)} + Q^{(1)} + R^{(2)} - R^{(3)} + Q^{(5)})X$

**Fig. 5.** Double-size modular multiplication algorithm A3

and

$$\begin{aligned} XA_1B_1 &\equiv_N X(Q^{(3)}X + R^{(3)}) \equiv_N Q^{(3)}\alpha + R^{(3)}X \; , \\ X^2A_1B_1 &\equiv_N X(R^{(3)}X + Q^{(3)}\alpha) \equiv_N R^{(3)}\alpha + Q^{(3)}\alpha X \; , \\ X(X-1)A_1B_1 &\equiv_N (-Q^{(3)} + R^{(3)})\alpha + (-R^{(3)} + Q^{(3)}\alpha)X \; , \end{aligned}$$

where-from

$$\begin{aligned} AB &\equiv_N \alpha(-Q^{(1)} + Q^{(2)} - Q^{(3)} + R^{(3)}) + R^{(1)} \\ &\qquad\qquad + X(-R^{(1)} + Q^{(1)} + R^{(2)} - R^{(3)} + Q^{(3)}\alpha) \\ &\equiv_N \alpha(-Q^{(1)} + Q^{(2)} - Q^{(3)} + R^{(3)}) + R^{(1)} \\ &\qquad\qquad + X(Q^{(4)}X + R^{(4)} - R^{(1)} + Q^{(1)} + R^{(2)} - R^{(3)}) \\ &\equiv_N \alpha(-Q^{(1)} + Q^{(2)} - Q^{(3)} + Q^{(4)} + R^{(3)}) + R^{(1)} \\ &\qquad\qquad + X(R^{(4)} - R^{(1)} + Q^{(1)} + R^{(2)} - R^{(3)}) \\ &\equiv_N (R^{(5)} + R^{(1)}) + (R^{(4)} - R^{(1)} + Q^{(1)} + R^{(2)} - R^{(3)} + Q^{(5)})X \; , \end{aligned}$$

which proves the correctness of A3. □

Again, this algorithm uses only 5 `MultModDiv` instructions. But if we take a careful look at its description, we observe that a couple of `MultModDiv` instructions are performed directly with operand $\alpha$. Therefore, having a small value for $\alpha$ would render these two `MultModDiv` instructions significantly faster. Suppose for example that an $n$-bit $X$ can be found given $N$ such that $\alpha \leq 2^{n/2}$. Assuming that the execution time of `MultModDiv` is essentially linear in the bitsize of its first operand, then Algorithm A3 would have a time consumption close to 4 `MultModDiv`, resulting in an additional speedup of 20%.

### 4.2 Extreme Cases: Down to 3 `MultModDiv`

Optimal performances are reached when $\alpha = -1, 2, 3$ for instance, in which cases the computational cost of our algorithm reduces to 3 `MultModDiv` instructions.

One may of course ask under which circumstances there exists an $n$-bit integer $X$ with such a trivial square modulo a $2n$-bit RSA modulus $N$. A practical way to ensure this consists in modifying the RSA key generation. We believe that simple algebraic techniques allow to do that while preserving the security of RSA moduli.

Other choices for the representation base may also present interesting properties, as we now illustrate. Assume for instance that for a given $N$, there exists an $n$-bit $Y \geq \lceil \sqrt{N} \rceil$ such that

$$Y^2 \equiv \alpha + \delta Y \pmod N ,$$

where we try to make $\alpha$ and $\delta$ as trivial as possible. If $\alpha$ and $\delta$ are simple numbers (ideally $\delta = 1$), A3 simplifies into Algorithm A4 depicted on Fig. 6.

| | |
|---|---|
| **Input:** | radix base $Y$, $2n$-bit integers $A = A_1Y + A_0, B = B_1Y + B_0$ |
| **Output:** | $AB \pmod N$ |
| **Cost:** | 3 `MultModDiv` |

$(Q^{(1)}, R^{(1)}) = \texttt{MultModDiv}(A_0, B_0, Y)$
$(Q^{(2)}, R^{(2)}) = \texttt{MultModDiv}(A_1 + A_0, B_1 + B_0, Y)$
$(Q^{(3)}, R^{(3)}) = \texttt{MultModDiv}(A_1, B_1, Y)$
Return $\alpha(-Q^{(1)} + Q^{(2)} - Q^{(3)} + R^{(3)} + Q^{(3)}\delta) + R^{(1)}$
$+ Y(-R^{(1)} - R^{(3)} + Q^{(1)} + R^{(2)} + Q^{(3)}(\alpha + \delta^2) + (-Q^{(3)} + R^{(3)} - Q^{(1)} + Q^{(2)})\delta)$

**Fig. 6.** Double-size modular multiplication algorithm A4

*Proof (of correctness for* A4*).* Using $Y^2 \equiv_N \alpha + \delta Y$ and Lemma 1, one gets

$$\begin{aligned}(Y-1)A_0B_0 &\equiv_N (Y-1)(Q^{(1)}Y + R^{(1)}) \\ &\equiv_N -Q^{(1)}Y - R^{(1)} + R^{(1)}Y + Q^{(1)}(\alpha + \delta Y) \\ &\equiv_N Q^{(1)}\alpha - R^{(1)} + (R^{(1)} - Q^{(1)} + Q^{(1)}\delta)Y ,\end{aligned}$$

$$\begin{aligned}Y(A_1 + A_0)(B_1 + B_0) &\equiv_N Y(Q^{(2)}Y + R^{(2)}) \\ &\equiv_N R^{(2)}Y + Q^{(2)}(\alpha + \delta Y) \\ &\equiv_N Q^{(2)}\alpha + (R^{(2)} + Q^{(2)}\delta)Y ,\end{aligned}$$

$$\begin{aligned}YA_1B_1 &\equiv_N Y(Q^{(3)}Y + R^{(3)}) \equiv_N Q^{(3)}\alpha + (R^{(3)} + Q^{(3)}\delta)Y \\ Y^2A_1B_1 &\equiv_N Y(Q^{(3)}\alpha + (R^{(3)} + Q^{(3)}\delta)Y) \\ &\equiv_N Q^{(3)}\alpha Y + (R^{(3)} + Q^{(3)}\delta)(\alpha + \delta Y) \\ &\equiv_N (R^{(3)} + Q^{(3)}\delta)\alpha + ((R^{(3)} + Q^{(3)}\delta)\delta + Q^{(3)}\alpha)Y ,\end{aligned}$$

$$\begin{aligned} Y(Y-1)A_1B_1 \equiv_N\ & (R^{(3)} - Q^{(3)} + Q^{(3)}\delta)\alpha \\ & + ((R^{(3)} + Q^{(3)}\delta)(\delta - 1) + Q^{(3)}\alpha)Y \\ \equiv_N\ & (R^{(3)} - Q^{(3)} + Q^{(3)}\delta)\alpha + (-R^{(3)} + (-Q^{(3)} + R^{(3)})\delta \\ & + Q^{(3)}(\delta^2 + \alpha))Y \ , \end{aligned}$$

so that

$$\begin{aligned} AB \equiv_N\ & \alpha(-Q^{(1)} + Q^{(2)} - Q^{(3)} + R^{(3)} + Q^{(3)}\delta) + R^{(1)} \\ & + Y(-R^{(1)} - R^{(3)} + Q^{(1)} + R^{(2)} + Q^{(3)}(\alpha + \delta^2) \\ & + (-Q^{(3)} + R^{(3)} - Q^{(1)} + Q^{(2)})\delta) \ , \end{aligned}$$

thereby proving Algorithm A4. □

Again, this algorithm has a cost of 3 `MultModDiv` instructions provided that the values for $\alpha$, $\delta$ and $\alpha+\delta^2$ are simple constant numbers. This could be ensured by properly adapting the RSA key generation algorithm.

## 5 Emulating Euclidean Multiplications

When the Euclidean multiplication itself is not directly available in hardware, it can be emulated easily with a couple of modular multiplications. The quotient of `MultModDiv` in [4] is calculated from the remainders of $x \cdot y$ modulo $z$ and modulo $(z+1)$. However, such a situation is most unfortunate for fast modular multiplication algorithms based on Montgomery's technique as either $z$ or $z+1$ is even. Although extensions of Montgomery to even moduli exist, we suggest a simple alternative hereafter. Our method is based on the next lemma.

**Lemma 2.** *If $0 < xy \le (z-1)^2$ then*

$$\left\lfloor \frac{xy}{z+\beta} \right\rfloor \le \left\lfloor \frac{xy}{z} \right\rfloor \le \left\lfloor \frac{xy}{z+\beta} \right\rfloor + \beta$$

*for any nonnegative $\beta$.*

*Proof.* Since $z < z + \beta$, it follows that $xy/(z+\beta) < xy/z$ and consequently $\lfloor xy/(z+\beta) \rfloor \le \lfloor xy/z \rfloor$. For the second inequality, we observe that

$$\frac{xy}{z} = \frac{xy}{z+\beta}\left(1 + \frac{\beta}{z}\right) \le \frac{xy}{z+\beta} + \frac{(z-1)^2\beta}{(z+\beta)z} < \frac{xy}{z+\beta} + \beta \ .$$

Therefore, we get $\lfloor xy/z \rfloor \le \lfloor xy/(z+\beta) \rfloor + \lceil \beta \rceil = \lfloor xy/(z+\beta) \rfloor + \beta$. □

So, letting

$$\Delta_\beta = \left\lfloor \frac{xy}{z} \right\rfloor - \left\lfloor \frac{xy}{z+\beta} \right\rfloor \quad \text{and} \quad C_\beta = xy \bmod (z+\beta) \ ,$$

(with $0 \le \Delta_\beta \le \beta$ by Lemma 1), one expresses the integer quotient resulting from the modular multiplication, $C = xy \bmod z$, as

$$\left\lfloor \frac{xy}{z} \right\rfloor = \frac{C - C_\beta - \Delta_\beta(z+\beta)}{\beta} \ . \tag{3}$$

*Proof.* By definition, we have $xy = \lfloor xy/z \rfloor z + C = \lfloor xy/(z+\beta) \rfloor (z+\beta) + C_\beta = (\lfloor xy/z \rfloor + \Delta_\beta)(z+\beta) + C_\beta$, which implies $C = \lfloor xy/z \rfloor \beta + \Delta_\beta (z+\beta) + C_\beta$. □

In particular, the value $\beta = 2$ yields the integer quotient from two modular reductions with moduli having the same parity as $z$. Carrying out a division by $\beta$ is inexpensive as it amounts to a shift of a single bit to the right. Finally, since $\Delta_2 \leq 2$, there are (at most) only two negligible corrections to make to get the exact value of the quotient.

*Remark 1.* This method readily extends for any value of $\beta$; the powers of 2 are of particular interest. Note also that a way to lower the expected error (cf. $\Delta_\beta$) consists in increasing the numerator in $\lfloor xy/(z+\beta) \rfloor$.

## 6 A Modular Multiplication Based Algorithm

When Euclidean multiplications are emulated in software from modular multiplications, one may wonder if using these directly could yield faster algorithms without necessarily coming back to RNS-based approaches [1, 2, 7, 9]. In this section, we propose a simple alternative to these works that keeps numbers under a radix representation. We rely on the following lemma.

**Lemma 3.** *Let $X$ be an $n$-bit odd integer not divisible by 3 and $N$ an arbitrary integer such that $X > \lceil \sqrt{N} \rceil$. There exists an algorithm which, given any $A = A_1X + A_0$ and $B = B_1X + B_0$ such that $A, B < N$ outputs the representation*

$$AB = C_3X^3 + C_2X^2 + C_1X + C_0$$

*in at most four $n$-bit modular multiplications.*

We refer the reader to Appendix A for a description of such an algorithm, which we denote by `Coefficients` in the sequel.

Now, very much in the spirit of Section 4.1, we precompute $X$ such that $X > \lceil \sqrt{N} \rceil$ and set $\alpha = X^2 \bmod N$. Here however, as we need $\gcd(X, 6) = 1$, we try out $X = \lceil \sqrt{kN} \rceil$ for increasing values of $k = 1, \ldots,$ until $X$ is found odd and coprime to 3. Even if $|X|$ exceeds $n$, the difference $|X| - n$ will be a very small value in any case, and we refer to the fact that we are able to work with non-reduced numbers when they do not exceed their range too much (see Section 2). Relying on Lemma 3, we devise Algorithm A5 as shown on Fig. 7.

*Proof (of correctness for* A5*).* By definition,

$$\begin{aligned}
AB &= U^{(1)}X^3 + V^{(1)}X^2 + W^{(1)}X + R^{(1)} \\
&\equiv_N R^{(1)} + V^{(1)}\alpha + (U^{(1)}\alpha + W^{(1)})X \\
&\equiv_N R^{(1)} + V^{(1)}\alpha + (Q^{(2)}X + R^{(2)} + W^{(1)})X \\
&\equiv_N R^{(1)} + (V^{(1)} + Q^{(2)})\alpha + (R^{(2)} + W^{(1)})X \\
&\equiv_N R^{(1)} + R^{(3)} + (R^{(2)} + W^{(1)} + Q^{(3)})X \ ,
\end{aligned}$$

thereby validating A5. □

| | |
|---|---|
| **Input:** | radix base $X$, $2n$-bit integers $A = A_1X + A_0, B = B_1X + B_0$ |
| **Output:** | $AB \pmod N$ |
| **Cost:** | 2 `MultModDiv` + 1 `Coefficients` |

$(U^{(1)}, V^{(1)}, W^{(1)}, R^{(1)}) = \texttt{Coefficients}(A, B, X)$
$(Q^{(2)}, R^{(2)}) = \texttt{MultModDiv}(\alpha, U^{(1)}, X)$
$(Q^{(3)}, R^{(3)}) = \texttt{MultModDiv}(\alpha, V^{(1)} + Q^{(2)}, X)$
Return $R^{(1)} + R^{(3)} + (R^{(2)} + W^{(1)} + Q^{(3)})X$

**Fig. 7.** Double-size modular multiplication algorithm A5

As indicated, our algorithm runs two `MultModDiv` and one `Coefficients` operations, which (relying on Section 5 or [4]) yields 8 modular multiplications among which 4 are executed with operand $\alpha$. Then, we can combine A5 with a proper modification of the RSA key generator to ensure that $\alpha$ is some small (absolute) constant. In this context of use, the cost of a double size modular multiplication by A5 reduces to four $n$-bit multiplications only, *i.e.*, becomes computationally equivalent to 2 calls to `MultModDiv` thereby yielding a speedup factor of $(14 - 4)/14 \approx 71\%$ in comparison with FS1.

## 7 Conclusion

In this paper, we showed how to optimally reduce the cost of Fischer and Seifert double-size modular multiplications, provided that the same basic operation (Euclidean multiplication) is available. We highlighted the role of the data representation towards the performance of emulated multiplications and proposed new ones featuring dramatic cost savings.

**Table 1.** Number of calls in double-size modular multiplication algorithms. The last line displays the number of equivalent $n$-bit modular multiplications

| **Calls** | Fischer-Seifert | | Our algorithms | | | | |
|---|---|---|---|---|---|---|---|
| | FS1 | FS2 | A1 | A2 | A3 | A4 | A5 |
| `MultModDiv` | 7 | 5 | 6 | 4 | $5 \to 3$ | 3 | $2 \to 0$ |
| `MultModDivInit` | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| `Coefficients` | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Equiv. `MultMod` | 14 | 12 | 12 | 10 | $10 \to 6$ | 6 | $8 \to 4$ |

We stress that in each and every of our algorithms, modifications of the radix base can be carried out while maintaining inputs and outputs under the same arithmetic format, which allows repeated executions with the same modulus. Naturally, the same algorithms may readily be used to perform double-size modular squarings. In the most favorable case, we emulate a double-size modular multiplication with no more than 3 Euclidean multiplications, resulting in a speedup factor of 57% in comparison with Fischer and Seifert's original procedures, as indicated in Table 1. When Euclidean multiplications cannot be carried out in hardware, we provide a variation based on modular multiplications only which surpasses original algorithms in performance by 71%. Although we doubt the existence of more advanced yet simple techniques, we challenge the cryptographic community for better results.

## References


1. J.-C. Bajard, L.-S. Didier, and P. Kornerup. An RNS Montgomery multiplication algorithm. In *13th IEEE Symposium on Computer Arithmetic (ARITH 13)*, pp. 234–239, IEEE Press, 1997.
2. J.-C. Bajard, L.-S. Didier, and P. Kornerup. An RNS Montgomery multiplication algorithm. *IEEE Transactions on Computers*, vol. 47, no. 7, pp. 766–776, 1998.
3. P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processing. In A.M. Odlyzko, Ed., *Advances in Cryptology – CRYPTO '86*, vol. 263 of *Lecture Notes in Computer Science*, pp. 311–323, Springer-Verlag, 1987.
4. W. Fischer and J.-P. Seifert. Increasing the bitlength of crypto-coprocessors via smart hardware/software co-design. In B.S. Kaliski Jr., Ç.K. Koç, and C. Paar, Eds., *Cryptographic Hardware and Embedded Systems – CHES 2002*, vol. 2523 of *Lecture Notes in Computer Science*, pp. 71–81, Springer-Verlag, 2003.
5. P.L. Montgomery. Modular multiplication without trial divisions. *Mathematics of Computations*, vol. 44, no. 170, pp. 519–521, 1985.
6. P. Paillier. Low-cost double-size modular exponentiation or how to stretch your cryptoprocessor. In H. Imai and Y. Zheng, Eds., *Public-Key Cryptography*, vol. 1560 of *Lecture Notes in Computer Science*, pp. 223–234, Springer-Verlag, 1999.
7. K.C. Posh and R. Posh. Modulo reduction in Residue Number Systems. *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 5, pp. 449–454, 1995.
8. J.-J. Quisquater. Fast modular exponentiation without division. Rump session of EUROCRYPT '90, Århus, Denmark, 1990.
9. J. Schwemmlein, K.C. Posh and R. Posh. RNS modulo reduction upon a restricted base value set and its applicability to RSA cryptography. *Computer & Security*, vol. 17, no. 7, pp. 637–650, 1998.
10. C.D. Walter. Faster modular multiplication by operand scaling. In J. Feigenbaum, Ed., *Advances in Cryptology – CRYPTO '91*, vol. 576 of *Lecture Notes in Computer Science*, pp. 313–323, Springer-Verlag, 1992.


## A Proof of Lemma 3

Our four modular multiplications will be

$$\begin{cases} R_0 &= (A \bmod X)(B \bmod X) \bmod X \ , \\ R_1 &= (A \bmod (X+1))(B \bmod (X+1)) \bmod (X+1) \ , \\ R_2 &= (A \bmod (X+2))(B \bmod (X+2)) \bmod (X+2) \ , \\ R_3 &= (A \bmod (2X+3))(B \bmod (2X+3)) \bmod (2X+3) \ . \end{cases}$$

In what follows, we use the notations

$$k_0 = AB \operatorname{div} X \ , \qquad k_1 = AB \operatorname{div} (X+1) \ ,$$
$$k_2 = AB \operatorname{div} (X+2) \ , \qquad k_3 = AB \operatorname{div} (2X+3) \ ,$$

and we have by definition

$$\begin{aligned} AB &= k_0 X + R_0 = k_1(X+1) + R_1 = k_0(X+1) + (R_0 - k_0) \\ &= k_2(X+2) + R_2 = k_0(X+2) + (R_0 - 2k_0) \ , \\ 2AB &= 2k_0 X + 2R_0 = 2k_3(2X+3) + 2R_3 = k_0(2X+3) - 3k_0 + 2R_0 \ , \end{aligned}$$

so that

$$\begin{aligned} k_0 &\equiv R_0 - R_1 \pmod{(X+1)} \\ 2k_0 &\equiv R_0 - R_2 \pmod{(X+2)} \\ 3k_0 &\equiv 2(R_0 - R_3) \pmod{(2X+3)} \ . \end{aligned}$$

Since $X$ is coprime to 3, if we call $a = (R_0 - R_2 + ((R_0 - R_2) \bmod 2)(X+2))/2 \bmod X+2$, $b = R_0 - R_1 \bmod (X+1)$ and $c = (2(R_0 - R_3) + (2(R_0 - R_3) \bmod 3)(2X+3))/3 \bmod (2X+3)$, we get that $k_0 \equiv b \pmod{(X+1)}$, $k_0 \equiv a \pmod{(X+2)}$ and $k_0 \equiv c \pmod{(2X+3)}$. Starting from these equations, we can perform Chinese remaindering, because $X+1$, $X+2$ and $2X+3$ are pairwise relatively prime:

$$k_0 \bmod ((X+1)(X+2)) = ((b-a) \bmod (X+1))(X+2) + a \ .$$

Letting $d = ((b-a) \bmod (X+1))$ and $e = a + 2d$, we have

$$k_0 \bmod ((X+1)(X+2)) = dX + e \ .$$

Moreover, remarking that $(X+1)(X+2)(-4) \equiv 1 \pmod{(2X+3)}$ and letting $f = -6d + 4e - 4c \bmod (2X+3)$, we notice that the second CRT recombination

$$\begin{aligned} k_0 &= [-4(c - dX - e) \bmod (2X+3)](X+1)(X+2) + dX + e \\ &= [2d(2X+3) - 6d + 4e - 4c) \bmod (2X+3)](X+1)(X+2) + dX + e \\ &= f(X+1)(X+2) + dX + e \end{aligned}$$

is easily rewritten as $k_0 = fX^2 + (d+3f)X + (e+2f)$. Consequently, $C = AB$ is computed in 4 modular multiplications as $C = C_3X^3 + C_2X^2 + C_1X + C_0$ with

$$\begin{aligned} C_3 &= f \ , \\ C_2 &= d + 3f \ , \\ C_1 &= e + 2f \ , \\ C_0 &= R_0 \ . \end{aligned}$$

Note that these operations are not of size $2n \times 2n$ modulo $n$ but of size $n \times n$ modulo $n$, because, from $A = A_1X + A_0$ and $B = B_1X + B_0$, $R_0$, $R_1$, $R_2$ and $R_3$ can be computed as

$$\begin{aligned}
R_0 &= A_0B_0 \bmod X \ , \\
R_1 &= (A_0 - A_1)(B_0 - B_1) \bmod (X+1) \ , \\
R_2 &= (A_0 - 2A_1)(B_0 - 2B_1) \bmod (X+2) \ , \\
R_3 &= (A_0 + (A_1 \bmod 2)X - 3(A_1 \operatorname{div} 2)) \\
&\qquad \times (B_0 + (B_1 \bmod 2)X - 3(B_1 \operatorname{div} 2)) \bmod (2X+3) \ .
\end{aligned}$$

As before, the cost of auxiliary operations (additions, subtractions, parity bits, etc.) is neglected. □