

# Applications of Fast Truncated Multiplication in Cryptography

Laszlo Hars

Seagate Research, 1251 Waterfront Place  
Pittsburgh, PA 15222, USA

[Laszlo@Hars.US](mailto:Laszlo@Hars.US)

**Abstract.** *Truncated Multiplications* compute *Truncated Products*, contiguous subsequences of the digits of the products of integers. They are based on the  $n$ -digit full multiplication algorithms of time complexity  $O(n^\alpha)$ , with  $1 < \alpha \leq 2$ , but a constant times faster. Applying these fast truncated multiplications several improved cryptographic long integer arithmetic algorithms are presented, including integer reciprocals, divisions, Barrett- and Montgomery- multiplications,  $2n$ -digit modular multiplication on HW for  $n$ -digit half products. E.g., Montgomery multiplication is performed in 2.6 Karatsuba multiplications time.

**Keywords:** *Computer Arithmetic, Short product, Truncated product, Cryptography, RSA cryptosystem, Modular multiplication, Montgomery multiplication, Karatsuba multiplication, Barrett multiplication, Approximate reciprocal, Optimization*

## 1 Notations

- Long integers are denoted by  $A = \{a_{n-1} \dots a_1, a_0\} = a_{n-1} \dots a_0 = \sum d^i a_i$  in a  $d$ -ary number system, where  $a_i, 0 \leq a_i \leq d-1$  are *digits* (usually 16 or 32 bits:  $d = 2^{16}$  or  $2^{32}$ )
- $|A|$  or  $|A|_d$  denotes the number of digits, the length of a  $d$ -ary number.  $|\{a_{n-1} \dots a_1 a_0\}| = n$
- $A \parallel B$  the number of the joined digit-sequence  $\{a_{n-1} \dots a_0 b_{m-1} \dots b_0\}$ ;  $|A| = n, |B| = m$
- $\lfloor x \rfloor$  denotes the integer part (floor) of  $x$ , and  $0 \leq \{x\} < 1$  is the fractional part, such that  $x = \lfloor x \rfloor + \{x\}$
- $\lg n = \log_2 n = \log n / \log 2$
- LS stands for **Least Significant**, the low order bit/s or digit/s of a number
- MS stands for **Most Significant**, the high order bit/s or digit/s of a number
- (*Grammar*) *School multiplication, division*: the digit-by-digit multiplication and division algorithms, as taught in elementary schools
- $A \times B, A \bowtie B$  denote the MS or LS half of the digit-sequence of  $A \times B$  (or  $A \cdot B$ ), respectively
- $A \otimes B$  denotes the middle third of the digit-sequence of  $A \times B$
- $M_\alpha(n)$  the time complexity of the Toom-Cook type full multiplication,  $M_\alpha(n) = O(n^\alpha)$ , with  $1 < \alpha \leq 2$
- $\gamma_\alpha$  = the speedup factor of the half multiplication, relative to  $M_\alpha(n)$
- $\delta_\alpha$  = the speedup factor of the middle-third product, relative to  $M_\alpha(n)$

## 2 Introduction

Many cryptographic algorithms are based on modular arithmetic operations. The most time critical operation is multiplication. For example, exponentiation, the fundamental building block of RSA-, ElGamal- or Elliptic Curve - cryptosystems or the Diffie-Hellman key exchange protocol: [17], is performed by a chain of modular multiplications. For *modular reduction* division is used, which can be performed via multiplication with the *reciprocal* of the divisor, so fast reciprocal calculation is also important. *Modular multiplications* can be performed with reciprocals and regular multiplications, and in some of these calculations truncated products are sufficient.

We present new speedup techniques for these and other basic arithmetic operations. For operand sizes of cryptographic applications school multiplication is used the most often, requiring simple control structure. Speed improvements can be achieved with Karatsuba's method and the Toom-Cook 3- or 4-way multiplication, but the asymptotically faster algorithms are slower for these operand lengths: [9], [14]. In this paper we consider digit-serial multiplication algorithms of time complexity  $O(n^a)$ ,  $1 < a \leq 2$ , similar to microprocessor software, that is, no massive-parallel- or discrete Fourier transform based multiplications, which require different optimization methods: [3].

## 3 Truncated Products

*Truncated Multiplication* computes a *Truncated Product*, a contiguous subsequence of the digits of the product of 2 integers. If they consist of the LS or MS half of the digits, they are sometimes called short products or half products. These are the most often used truncated products together with the computation of the middle third of the product-digits, also called middle product.

No exact speedup factor is known for truncated multiplications, which are based on full multiplications faster than the school multiplication. For half products computed by Fourier transform based multiplications no constant time speedup is known. Fast truncated product algorithms are discussed and analyzed in the accompanying paper: *Fast Truncated Multiplication for Cryptographic Applications*.

## 4 Time complexity

Multiplication is more expensive (slower and/or more hardware consuming) even on single digits, than addition or store/load operations. Many computing platforms perform additive- and data movement operations parallel to multiplications, so they don't take extra time. In order to obtain general results and to avoid complications from architecture dependent constants we measure the time complexity of the algorithms with the *number of digit-multiplications* performed.

For the commonly used multiplication algorithms, even for moderate operand lengths the number of digit-multiplications is well approximated by  $n^a$ , where  $a$  is listed in the table below.

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
2	$\log^3/\log 2 = 1.5850$	$\log^5/\log 3 = 1.4650$	$\log^7/\log 4 = 1.4037$

On shorter operands asymptotically slower algorithms could be faster, when architecture dependent minor terms are not yet negligible. (We cannot compare different multiplication algorithms, running in different computing environments, without knowing all these factors.) For example, when multiplying linear combinations of partial results or operands, a significant number of non-multiplicative digit operations are executed, that might not be possible to perform in parallel to the digit-multiplications. They affect some minor terms in the complexity expressions and could affect the speed relations for shorter operands. To avoid this problem, when we look for speedups for certain multiplication algorithms, when not all of their product digits are needed, we only consider algorithms performing *no more auxiliary digit operations than what the corresponding full multiplication performs*. When each member of a family of algorithms under this assumption uses internally one kind of black-box multiplication method (School, Karatsuba, Toom-Cook- $k$ ), the speed ratios among them are about the same as that of the black-box multiplications. Consequently, if on a given computational platform and operand length one particular multiplication algorithm is found to be the best, say it is Karatsuba, then, within a small margin, the fastest algorithm discussed in this paper is also the one, which uses Karatsuba multiplication.

## 5 Reciprocal

At calculating  $1/x$  it is convenient to treat the  $n$ -digit integer  $x$ , as a binary fixed point number, assuming the binary point in front of the first nonzero bit ( $0.5 \leq x < 1$ ) and scale (shift) the result after the reciprocal calculations to get the integer reciprocal  $\mu = \lfloor d^{2^n}/x \rfloor$ .

Newton's iteration is a very fast algorithm for computing reciprocals. It starts with a suitable initial estimate of the reciprocal, which can be read from a look-up table or calculated with 6 digit-multiplications and 5 additions as shown in Figure 1, with constants in the innermost parentheses. On sufficiently precise arithmetic engines it provides more than 34 bit accurate initial estimate  $r$  of  $1/x$ .

$$\begin{aligned}
\mathbf{r} &= 2.91421 - 2 \cdot \mathbf{x} \\
\mathbf{r} &= \mathbf{r} \cdot ((2 + 1.926 \cdot 2^{-09}) - \mathbf{x} \cdot \mathbf{r}) \\
\mathbf{r} &= \mathbf{r} \cdot ((2 + 1.926 \cdot 2^{-18}) - \mathbf{x} \cdot \mathbf{r}) \\
\mathbf{r} &= \mathbf{r} \cdot ((2 + 1.530 \cdot 2^{-36}) - \mathbf{x} \cdot \mathbf{r})
\end{aligned}$$

**Figure 1.** 34.5-bit initial reciprocal

Each *Newton iteration* of  $r \leftarrow r(2-rx)$  doubles the number of accurate digits. With initial error  $\varepsilon$

$$r = \frac{1}{x}(1-\varepsilon); \quad r \leftarrow \frac{1}{x}(1-\varepsilon)(2-(1-\varepsilon)) = \frac{1}{x}(1-\varepsilon^2).$$

If started with 32 bit accuracy, the iterations give approximate reciprocal values of  $k = 64, 128, 256 \dots$  bit accuracy. The newly calculated  $r$  values are always rounded to  $k$  digits, and the multiplications, which computed them, need not be more than  $k$ -digit accurate. Some work can be saved by arranging the calculations according to the modified recurrence expression  $r \leftarrow 2r + r^2(-x)$ . The most significant digits of  $r$  don't change, so we just calculate the necessary new digits and attach them to  $r$ :

$$r_{k+1} = r_k \parallel \text{digits}[2^k + 1 \dots 2^{k+1} \text{ of } r^2(-x)].$$

Having an  $m = 2^k$ -digit accurate reciprocal  $r_k$  we perform a  $m$ -digit squaring ( $m \cdot (m+1)/2$  steps with school multiplication) and a  $2m \times 2m$  multiplication with the result- and  $2m$  digits of  $-x$ . Only the digits  $m+1 \dots 2m$  have to be calculated. This is a *third-quarter* product. With school multiplication it takes  $1/2 m^2$  digit-products. Together with the  $m$ -digit squaring it is  $2m^2 + O(m)$  steps. Summing these up, for  $n$ -digit accuracy, the time complexity is  $R_2(n) = 2(1+2^2+4^2+\dots+(n/2)^2) = \frac{2}{3}n^2 - \frac{2}{3}$ . However, there is a better way to organize the work:

**Algorithm R.** Arrange the calculation according to:  $r_{k+1} \leftarrow r_k + r_k(1-r_kx)$ . Here,  $r_kx \approx 1 - d^{-2k}$  ( $2^k$ -digit accuracy, if we started with 1 accurate digit approximation), so the  $m = 2^k$  MS digits of  $r_kx$  are all  $d-1$ , they need not be calculated.

We use  $2m$  digits of  $-x$ , instead of  $x$ , but only the middle  $m$  digits of the  $3m$ -digit long product are needed (*middle third* product). The result is multiplied with  $r$ , but only the MS  $m$  digits are interesting (the first multiplicand is shifted), which is an MS half product. It is still a shifted result, so appending the new  $m$  digits to the previous approximation gives the new one (with the notation  $-x_{(2m)} := \text{MS}_{2m}(d^n - x)$ ):

$$r_{k+1} = r_k \parallel r_k \times (r_k \otimes -x_{(2m)}).$$

The series of multiplications take  $\boxed{(\delta_\alpha + \gamma_\alpha) \sum_{k=1,2,4,\dots,n/2} M_\alpha(k)}$  time, summing up to the following ratios compared to the corresponding multiplication time  $M_\alpha(n)$ :

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
0.5	0.9039	<b>1.4379</b>	<b>1.5927</b>

**Note 1.** There are no other digit-operations in this algorithm than multiplications and load/stores (and the initial negation of  $x$ , if no parallel digit multiply-subtract operation is available). Therefore, it is conform to our complexity requirements (fewer auxiliary operations than at multiplications).

We have left out all of the details with the rounding (see [28]). One needs to keep some guard digits with  $b$  accurate bits. These would increase to  $2b$  accurate guard bits at the next iteration, but the rounding errors (omitted carries) destroy some of them. With the proper choice of  $b$  the rounding problems remain in the guard digits and the accuracy of the rest doubles at each Newton iteration.

The most important *results* are that  $n$ -digit accurate reciprocals can be calculated in half of the time of an  $n \times n$  -digit school multiplication, and 90% of a Karatsuba multiplication.

**Note 2.** The speedup in Algorithm R (concatenations instead of additions and the pre-calculation of  $-x$ ) are necessary to avoid large number of additions, forbidden in our complexity model. However, they only improve minor terms of the time complexity. For the Karatsuba case the main term (and so the asymptotic complexity of the reciprocal algorithm) is the same as in [11], the results for the Toom-Cook multiplications are new.

## 6 Long division

Newton's method calculates an approximate reciprocal of the divisor  $x$ . Multiplying the dividend  $y$  with it gives the quotient. (Another multiplication and subtraction gives the remainder. See more at Barrett's multiplication, below.) For cryptography the interesting case is  $2n$  digits long dividend, over  $n$ -digit divisor. The quotient is also  $n$  digits long, dependent on the MS digits of  $y$ . (Other length relations can be handled by cutting the dividend into pieces or padding it with 0's.)

The Karp-Markstein trick [14] incorporates the final multiplication ( $y \cdot 1/x$ ) into the last Newton iteration:

$$z_{n/2} \leftarrow r_{n/2} \times y_{2n-1 \dots 3n/2}; \lfloor y/x \rfloor = z_{n/2} \parallel r_{n/2} \times (y_{3n/2-1 \dots n} - z_{n/2} \otimes x).$$

The complexity of the final Newton iteration remains the same, but the multiplication step becomes faster:  $\gamma_a M_a(n/2)$ . It gives the complexity of calculating the quotient of a  $2n$ -digit dividend over an  $n$ -digit divisor (relative to  $M_a(n)$ ):

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
0.625	1.1732	<b>1.7596</b>	<b>1.9416</b>

With school multiplication the division is significantly faster than multiplication (but only half as many digits are computed). With Karatsuba multiplication it is only 17% slower (at practical operand lengths the coefficient is closer to 1.5: [28]), and the most common Toom-Cook divisions are still faster than 2 multiplications. The source of this speedup is the dissection of the operands and working on individual digit-blocks, making use of the algebraic structure of the division.

In cryptographic applications many divisions are performed with the same divisor. In this case, the time for calculating the reciprocal becomes negligible compared to the time of the many multiplications, so the amortized cost of the division is one *half multiplication*:  $\boxed{\gamma_a M_a(n)}$ .

**Historical note.** In [11] the Karatsuba case was analyzed and also a faster direct division algorithm was presented, which reduces the coefficient for the Karatsuba division to 1. Unfortunately, the direct division algorithm needs complicated carry-save techniques, which increases the number of auxiliary operations beyond the limit of our complexity model. In [6] practical direct division with Karatsuba complexity was presented. Its empirical complexity coefficient was around 2 on a particular computer, but that includes all the non-multiplicative operations, so we cannot directly compare it to the results here.

## 7 Barrett Multiplication

**Algorithm B.** Modular multiplications are calculated with

$$ab \bmod m = ab - \lfloor ab/m \rfloor m = \text{LS}(ab) - (\text{MS}(ab) \times \mu) \times m \text{ with } \mu = \lfloor d^{2n}/m \rfloor$$

To take advantage of the unchanging modulus,  $\mu = 1/m$  is calculated beforehand to multiply with. It is scaled to make it suitable for integer arithmetic, that is,  $\mu = \lfloor d^{2n}/m \rfloor$  is calculated ( $n$  digits and 1 bit long). Multiplying with that and keeping the most significant  $n$  bits only, the error is at most 2, compared to the exact division. The MS digits of  $ab$  and  $\lfloor ab/m \rfloor m$  are the same, so only the LS  $n$  digits of both are needed. These yield the algorithm given in Figure 2. There, too, the truncated products can be calculated faster than the full products.

```

(A1dn + A0) ← a × b
q ← A1 × μ
r ← A0 - q × m
if r < 0: r ← r + dn+1
while r ≥ m: r ← r - m

```

**Figure 2.** Barrett's multiplication

In practice a few extra bits precision is needed to guarantee that the last "while" loop does not cycle many times. This increase in length of the operands makes the algorithm with school multiplications slightly slower than the Montgomery multiplication [4]. Also,  $\mu$  and  $q$  requires  $2n$  digits extra memory. On the other hand, the advantage of this method is that it can be directly applied to modular reduction of messages in crypto applications; there is no need to transform data to special form and adapt algorithms. The pre-computation is simple and fast (taking less than half of the time of one modular multiplication in case of school- or Karatsuba multiplication).

The dominant part of the time complexity of Barrett's multiplication is  $\boxed{(1+2\gamma_a)M_a(n)}$ , a significant improvement over the previous best results of 3 (2 for the school multiplication) in [6]. The speed ratios over  $M_a(n)$  are tabulated below:

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
2	<b>2.6155</b>	<b>2.7762</b>	<b>2.8464</b>

**Modular Squaring.** Since the non-modular square  $a^2$  is calculated almost twice faster than the general products  $ab$ , the first step of the Barrett multiplication becomes faster. The rest of the algorithm is unchanged, giving the speed ratio  $\boxed{0.5+2\gamma_a}$  of modular squaring over the  $n \times n$ -digit multiplication time  $M_a(n)$ :

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
1.5	<b>2.1155</b>	<b>2.2762</b>	<b>2.3464</b>

**Constant operand.** With pre-calculations we can speedup those Barrett multiplications, which have one operand constant. It is very important in long exponentiations computed with the binary- or multiply-square exponentiation algorithms, where the multiplications are either squares or performed with a constant (in the RSA case it is the message or ciphertext).

With  $b$  constant, one can pre-calculate the  $n$  digits long  $\beta' := MS_n(b/m)$ . With it

$$a \cdot b \bmod m = a \times b - (a \times \beta') \times m.$$

The corresponding algorithm runs faster, in  $3\gamma_a M_a(n) < (1 + 2\gamma_a)M_a(n)$  time. There is another way of expressing the modular multiplication, with fractional parts:  $ab \bmod m = \{ab/m\}m$ . This leads to an even faster algorithm:

**Algorithm BC.** Pre-calculate  $\beta := \text{MS}_{2n}(b/m)$ ,  $2n$  digits. The MS  $n$  digits of the fractional part  $\{ab/m\}$  is  $a \otimes \beta$ , so

$$a \cdot b \bmod m = (a \otimes \beta) \times m.$$

For Barrett multiplications with constants this equation gives speed ratios over  $M_a(n)$ :  $\frac{(\delta_a + \gamma_a)M_a(n)}{M_a(n)}$ . It is close to the squaring time, a significant *improvement* over previously used general multiplication algorithms.

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
1.5	<b>1.8078</b>	<b>2.5316</b>	<b>2.6211</b>

**Note.** Barrett's modular multiplication calculates the quotient  $q = \lfloor ab/m \rfloor$ , as well (line 2 in Figure 2). If it is needed, the final correction step (while-loop) has to increment its value, too.

## 8 Montgomery Multiplication

As originally formulated in [18] the Montgomery multiplication is of quadratic time, doing interleaved modular reductions, and so it could not take advantage of truncated products. It is simple and fast, performing a right-to-left division (also called exact division or odd division: [12]). In this direction there are no problems with carries (which propagate away from the processed digits) or with estimating the quotient digit wrong, so no correction steps are necessary. These give it some 6% speed advantage over the original Barrett's multiplication and 20% speed advantage over the direct division based reduction, using school multiplications: [4].

Montgomery's multiplication calculates the product in "row order", so a little tweaking is necessary for speeding it up at squaring [27]. The price for the simplicity of the modular reduction is that the multiplicands have to be converted to a special form before the calculations and back at the end, i.e., pre- and post-processing is necessary, each taking time comparable to a modular multiplication.

```

for i = 0..n-1
    t = xi · m' mod d
    x = x + t · m · di
x = x / dn
if ( x ≥ m )
    x = x - m

```

**Figure 3.** Montgomery reduction

In Figure 3 the Montgomery reduction is described. The single digit  $m' = -m_0^{-1} \bmod d$  is a pre-calculated constant, which exists if  $m$  is odd. (It is in cryptography, because  $m$  is a large prime or a product of large primes.)

The rationale behind the algorithm is representing a long integer  $a$ ,  $0 \leq a < m$ , as  $aR \bmod m$  with  $R = d^n$ . The modular product of 2 numbers in this representation is  $(aR)(bR) \bmod m$ , which is converted to the right form by multiplying with  $R^{-1}$ , since  $(aR)(bR)R^{-1} \bmod m = (ab)R \bmod m$ . This correction step,  $x \rightarrow xR^{-1} \bmod m$  is called the *Montgomery reduction*. The product  $AB$  can be calculated prior to the reduction ( $n$  digits of extra memory needed), or interleaved with the reduction. The later is called the *Montgomery multiplication* ( Figure 4 ).

```

x0 = 0
for i = 0..n-1
    t = (x0 + ai · b0) · m' mod d
    x = (x + ai · b + t · m) / d
if ( x ≥ m )
    x = x - m

```

**Figure 4.** Montgomery multiplication

### 8.1 Montgomery multiplications with Karatsuba complexity and faster

Montgomery's reduction implicitly finds  $u$  (the  $t$  values form its digits) for the  $2n$ -digit  $x$ , such that  $x + u \cdot m = z \cdot d^n$ , with an  $n$ -digit  $z$ , which is the result of the reduction. Taking this equation mod  $d^n$  we get:  $-x \equiv u \cdot m \bmod d^n$ , or  $u = x \cdot (-m^{-1}) \bmod d^n = x \times (-m^{-1})$ . (Here  $-m^{-1} \bmod d^n$  can be pre-calculated with a modular inverse algorithm.) These prove the validity of the following

**Algorithm M.** Montgomery reduction:  $x d^{-n} \bmod m = MS(x) - (LS(x) \times (-m^{-1})) \times m$ .  
□

With 2 half products and one full multiplication (to get  $a \cdot b$ ) the above algorithm takes exactly as much time as the Barrett multiplication  $\boxed{(1 + 2\gamma_a)M_a(n)}$  (with the same squaring speedup as at Barrett):

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
2	<b>2.6155</b>	<b>2.7762</b>	<b>2.8464</b>

**Algorithm MC.** Montgomery multiplication with constants is calculated in  $\lceil 3 \gamma_a M_a(n) \rceil$  time with:

$$\beta := b \times (-m^{-1}), \quad a b d^{-n} \bmod m = a \times b - (a \times \beta) \times m.$$

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
<b>1.5</b>	<b>2.4233</b>	<b>2.6644</b>	<b>2.7696</b>

**Note.** These are new, faster algorithms for the Montgomery multiplication using Karatsuba or Toom-Cook multiplications. However, the advantage of the simplicity of the right-to-left division is lost, but the high costs of pre- and post processing remain. Therefore, unless some other part of the system requires the result in Montgomery form, there is no reason for using Montgomery multiplications until a significant speedup is found. With sub-quadratic multiplication algorithms currently Barrett's method is faster. If storage space is a concern, direct division based modular reductions work better, not needing pre-computation or extra memory: [6], [11].

## 9 Quadruple length modular multiplication

Below two algorithms are presented for modular multiplication of  $2n$  bit numbers using truncated  $n$  bit arithmetic (actually 3 bits more for guard digits and handling overflows). The reasons behind their design are also given, helping easy adaptation to arithmetic processors with different capabilities. They are useful, for example if a 512/1024-bit black-box secure coprocessor is used for RSA-2048 calculations. The presented algorithms are similar to the modular multiplication algorithm in [10] modified in [6], but our computational model is different. The main advantage of our algorithms is speed (very few calls to the coprocessor) and that the coprocessor can be very simple (only half and full multiplications and additions are needed).

The coprocessor performs  $n/2$ -by- $n/2$ -digit multiplications or  $n$ -by- $n$ -digit half multiplications returning  $n$ -digit results. The caller dis/assembles numbers from their parts, reads and writes at most  $n$ -digit numbers to/from the coprocessor's registers and requests (truncated) multiplications or additions of  $n$ -digit numbers. We saw earlier that with these instructions integer reciprocals can be calculated, and using them with the Barrett multiplication we computed the quotient  $\lfloor ab/m \rfloor$  and the remainder  $ab \bmod m$ . Of course, other modular multiplication and reduction algorithms work, too.

## 9.1 Algorithm Q1

Denote the  $2n$ -digit operands and their halves by

$$\begin{aligned} a &= \{a_1, a_0\} = a_1 d^n + a_0 \\ b &= \{b_1, b_0\} = b_1 d^n + b_0 \\ m &= \{m_1, m_0\} = m_1 d^n + m_0 \\ 0 &\leq a_1, a_0, b_1, b_0, m_1, m_0 < d. \end{aligned}$$

We assume that  $m$  is normalized, that is,  $d^{2n}/2 \leq m < d^{2n}$ . If not, replace it with the normalized  $2^k m$  and perform a modular reduction of the final result.

Let us split the middle partial products, allowing the caller to cut the full product into exact halves:

$$\begin{aligned} L &= \text{LS}(a_0 b_1) + \text{LS}(a_1 b_0) \\ M &= \text{MS}(a_0 b_1) + \text{MS}(a_1 b_0) \end{aligned}$$

The product  $a \cdot b$  can be expressed with them as  $d^n(d^n(a_1 b_1 + M) + L) + a_0 b_0$ . The modular reduction is performed by subtracting multiples of  $m$  from  $ab$ , until the result gets close to  $m$ . A few more times adding/subtracting  $m$  then finishes the job.

```

(q1, r1) = ModMult(a1, b1, m1)
M = a0 × b1 + a1 × b0
x1 = {q1 × m0, q1 × m0}
(q2, r2) = ModRed(dn(r1 + M) - x1, m1)
L = a0 × b1 + a1 × b0
x2 = {q2 × m0, q2 × m0}
c0 = {a0 × b0, a0 × b0}
R = dn(r2 + L) - x2 + c0
while (R ≥ m)
    R = R - m
while (R < 0)
    R = R + m

```

**Figure 5.** Q1 Quad-length modular multiplication

- The largest term  $a_1 b_1 d^{2n}$  is reduced by  $n$  digits with Modular Multiplication:  
 $(q_1, r_1) \leftarrow \text{ModMult}(a_1, b_1, m_1)$ .  
 Taking  $d^n q_1 m$  from  $ab$  cancels the MS  $n$  digits.
- $d^n(d^n(r_1 + M) + L - q_1 m_0) + a_0 b_0$  is left. Cancel the MS digit as before with modular reduction:  
 $(q_2, r_2) \leftarrow \text{ModRed}(d^n(r_1 + M) + L - q_1 m_0, m_1)$ .

Note that the 1<sup>st</sup> argument of ModRed is  $2n$  digits long, but we can process the MS and LS halves separately, like Barrett's algorithm does (see in Figure 2).

The modular reduction is actually subtracting  $q_2m$ . It leaves:

$$R = d^n(r_2+L) - q_2m_0 + a_0b_0.$$

Each product is at most  $2n$  digits long, so adding the modulus  $m$  to  $R$  or subtracting it from  $R$  at most 4 times reduces the result to  $0 \leq R < m$ .

**Proposition.** The above Algorithm Q1 computes the  $2n$ -digit  $(ab \bmod m)$  with at most 16 half multiplications of  $n$  digits and one (pre-computed)  $n$ -digit reciprocal.

**Proof.**  $R = ab - km$  for some integer  $k$ , and  $0 \leq R < m \Rightarrow R = ab \bmod m$ .

In Figure 5 there are 10 half products. If Barrett's modular multiplication is applied, it computes 4 half products, his reduction does 2, making it 16. Both moduli were the  $n$ -digit  $m_1$ .  $\square$

**Note.** When school multiplications are used to calculate the half products, Algorithm Q1 takes the time of roughly 8 normal multiplications, or 4 modular multiplications of  $n$ -digit numbers.

## 9.2 Algorithm Q2

The parameters are processed in halves, so it seems natural to use Karatsuba's trick to trade a couple of half products for additions. The middle terms are calculated with multiplying the differences of the MS and LS halves of the multiplicands, and combining the result with the LS and MS half products. They are also split, allowing the caller to build the full product from the halves:

$$\begin{aligned} L &= \text{LS}(a_0b_0) + \text{LS}(a_1b_1) - \text{LS}((a_1-a_0)(b_1-b_0)) \\ M &= \text{MS}(a_0b_0) + \text{MS}(a_1b_1) - \text{MS}((a_1-a_0)(b_1-b_0)) \end{aligned}$$

The product  $a \cdot b$  can still be expressed with them as  $d^n(d^n(a_1b_1+M)+L) + a_0b_0$ . The modular reduction is performed by subtracting multiples of  $m$ , until the result gets close to  $m$ , exactly as before at Algorithm Q1, except the modular multiplication can be replaced with modular reduction, since the product  $c_1 = a_1b_1$  has already been calculated.

```

c11 = a1 × b1; c10 = a1 × b0
c01 = a0 × b0; c00 = a0 × b0
M = c01 + c11 + (a1 - a0) × (b1 - b0)
L = c00 + c10 + (a1 - a0) × (b1 - b0)
(q1, r1) = ModRed(dnc11 + c10, m1)
x1 = {q1 × m0, q1 × m0}
(q2, r2) = ModRed(dn(r1 + M) - x1, m1)
x2 = {q2 × m0, q2 × m0}
R = dn(r2 + L) - x2 + c0
while (R ≥ m)
    R = R - m
while (R < 0)
    R = R + m

```

**Figure 6.** Q2 Quad-length modular multiplication

**Proposition.** Algorithm Q2 computes the  $2n$ -digit  $(ab \bmod m)$  with at most 14 half multiplications of  $n$  digits and one (pre-computed)  $n$ -digit reciprocal.

**Proof.**  $R = ab - km$  for some integer  $k$ , and  $0 \leq R < m \Rightarrow R = ab \bmod m$ . In Figure 6 there are 10 half products. If Barrett's reductions are applied, they calculate 2 half products, making it 14. These reductions were performed with modulus  $m_1$ , the MS half of the  $2n$ -digit modulus. (It helps reducing the pre-computation work, because the hidden constant  $\mu_1$  is also only  $n$  digits long.) []

**Note.** When school multiplications are used to calculate the half products, Algorithm Q2 takes the time of roughly 7 regular multiplications, or 3.5 modular multiplications of  $n$ -digit numbers.

## 10 Summary

General optimizations and the use of fast truncated multiplication algorithms allowed us to improve the performance of several cryptographic algorithms based on long integer arithmetic. The most important results presented in the paper:

- Fast initialization of the Newton reciprocal algorithm
- Fast Newton's reciprocal algorithm with only truncated product arithmetic (without external additions or subtractions)
- New long integer division algorithms based on Toom-Cook multiplications
- Accelerated Barrett multiplication with Karatsuba complexity and faster
- Speedup of Barrett's squaring and multiplication when one multiplicand is constant

- New algorithm for Montgomery multiplication with Karatsuba complexity and faster
- Speedup of Montgomery squaring and multiplication when one multiplicand is constant
- Fast and adaptable quad-length modular multiplications on short arithmetic co-processors

In practice a combinations of different algorithms is employed for multiplication. For example, Karatsuba multiplication is used until the recursion reduces the operand size below a certain threshold, like 8 digits. At that point school multiplication becomes faster, so it is used for shorter operands. The analysis of such hybrid methods depends on factors reflecting HW or SW features, constraints. *Because of space limitations our HW and SW implementation- and simulation results are collected in a separate paper* [28]. The reference implementations of the presented algorithms (for several versions of embedded processors) were sometimes slightly faster than expected in the hybrid cases, because of the larger speedups possible for short operands, sometimes somewhat slower, because of the rounded fractional splitting points. Also, the cost of calculating guard digits (for handling carry problems) could affect the speed relations of the algorithms around the boundaries. The results are very much dependent on the characteristics of the HW platforms (word length, parallel instructions, instruction timings, instruction pipeline, cache, virtual/paging memory, etc.).

## 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] P. D. Barrett, *Implementing the Rivest Shamir Adleman public key encryption algorithm on standard digital signal processor*, In Advances in Cryptology-Crypto'86, Springer-Verlag, 1987, pp.311-323.
- [3] D. J. Bernstein, *Fast Multiplication and its Applications*, <http://cr.yp.to/papers.html#multapps>
- [4] Bosselaers, R. Govaerts and J. Vandewalle, *Comparison of three modular reduction functions*, In Advances in Cryptology-Crypto'93, LNCS 773, Springer-Verlag, 1994, pp.175-186.
- [5] E.F. Brickell. *A Survey of Hardware Implementations of RSA*. Proceedings of Crypto'89, Lecture Notes in Computer Science, Springer-Verlag, 1990.
- [6] C. Burnikel, J. Ziegler, *Fast recursive division*, MPI research report I-98-1-022
- [7] B. Chevallier-Mames, M. Joye, and P. Paillier. *Faster Double-Size Modular Multiplication from Euclidean Multipliers*, Cryptographic Hardware and Embedded Systems – CHES 2003, vol. 2779 of Lecture Notes in Computer Science, pp. 214–227, Springer-Verlag, 2003.
- [8] J.-F. Dhem, J.-J. Quisquater, *Recent results on modular multiplications for smart cards*, Proceedings of Cardis 1998, Volume 1820 of Lecture Notes in Computer Security, pp 350-366, Springer-Verlag, 2000
- [9] GNU Multiple Precision Arithmetic Library manual <http://www.swox.com/gmp/gmp-man-4.1.2.pdf>

- [10] W. Fischer and J.-P. Seifert. *Increasing the bitlength of crypto-coprocessors via smart hardware/software co-design*. Cryptographic Hardware and Embedded Systems – CHES 2002, vol. 2523 of Lecture Notes in Computer Science, pp. 71–81, Springer-Verlag, 2002.
- [11] G. Hanrot, M. Quercia, P. Zimmermann, *The Middle Product Algorithm, I*. Rapport de recherche No. 4664, Dec 2, 2002 <http://www.inria.fr/rrrt/rr-4664.html>
- [12] K. Hensel, *Theorie der algebraische Zahlen*. Leipzig, 1908
- [13] J. Jedwab and C. J. Mitchell. *Minimum weight modified signed-digit representations and fast exponentiation*. Electronics Letters, 25(17):1171-1172, 17. August 1989.
- [14] A. H. Karp, P. Markstein. *High precision division and square root*. ACM Transaction on Mathematical Software, Vol. 23, n. 4, 1997, pp 561-589.
- [15] D. E. Knuth. *The Art of Computer Programming*. Volume 2. Seminumerical Algorithms. Addison-Wesley, 1981. Algorithm 4.3.3R
- [16] W. Krandick, J.R. Johnson, *Efficient Multiprecision Floating Point Multiplication with Exact Rounding*, Tech. Rep. 93-76, RISC-Linz, Johannes Kepler University, Linz, Austria, 1993.
- [17] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.
- [18] P.L. Montgomery, "Modular Multiplication without Trial Division," Mathematics of Computation, Vol. 44, No. 170, 1985, pp. 519-521.
- [19] T. Mulders. *On computing short products*. Tech Report No. 276, Dept of CS, ETH Zurich, Nov. 1997 <http://www.inf.ethz.ch/research/publications/data/tech-reports/2xx/276.pdf>
- [20] P. Paillier. *Low-cost double-size modular exponentiation or how to stretch your cryptoprocessor*. Public-Key Cryptography, vol. 1560 of Lecture Notes in Computer Science, pp. 223–234, Springer-Verlag, 1999.
- [21] 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.
- [22] J.-J. Quisquater, *Fast modular exponentiation without division*. Rump session of Eurocrypt'90, Aarhus, Denmark, 1990.
- [23] R. L. Rivest; A. Shamir, and L. Adleman. 1978. *A method for obtaining digital signatures and public key cryptosystems*. Communications of the ACM 21(2):120--126
- [24] J. Schwemmlin, 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.
- [25] SNIA OSD Technical Work Group [http://www.snia.org/tech\\_activities/workgroups/osd/](http://www.snia.org/tech_activities/workgroups/osd/)
- [26] C.D. Walter, "Faster modular multiplication by operand scaling," Advances in Cryptology, Proc. Crypto'91, LNCS 576, J. Feigenbaum, Ed., Springer-Verlag, 1992, pp. 313–323.
- [27] L. Hars, "Long Modular Multiplication for Cryptographic Applications", CHES 2004, Misprinted: LNCS 3156, pp 44-61. Springer-Verlag, 2004. <http://eprint.iacr.org/2004/198/>
- [28] L. Hars, "Finding the Fastest Multiplication for Cryptographic Operand Lengths: Analytic and Experimental Comparisons" manuscript.