Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 61721, 9 pages
doi:10.1155/2007/61721

Research Article

Applications of Fast Truncated Multiplication in Cryptography

Laszlo Hars

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

Received 30 June 2006; Revised 3 September 2006; Accepted 17 October 2006

Recommended by Sandro Bartolini

Truncated multiplications compute *truncated products*, contiguous subsequences of the digits of integer products. For an n-digit multiplication algorithm of time complexity $O(n^\alpha)$, with $1 < \alpha \leq 2$, there is a truncated multiplication algorithm, which is constant times faster when computing a short enough truncated product. Applying these fast truncated multiplications, several cryptographic long integer arithmetic algorithms are improved, including integer reciprocals, divisions, Barrett and Montgomery multiplications, $2n$-digit modular multiplication on hardware for n-digit half products. For example, Montgomery multiplication is performed in 2.6 Karatsuba multiplication time.

1. INTRODUCTION

Embedded systems, like cell phones, cable modems, wireless routers/modems, portable media players, DVD players, set-top TV boxes, digital VCRs, secure disk drives, FLASH memories, smart cards, cryptographic tokens, and so forth often use some forms of public key cryptography, employing long integer arithmetic. These devices are usually resource constrained, and so speed improvements of the used algorithms are important, allowing slower clocked processors, memory, and so reducing power consumption, heat dissipation, and costs.

Many of these cryptographic algorithms are based on modular arithmetic operations. The most time critical one is *modular multiplication*. Exponentiation is performed by a chain of these, and it is the fundamental building block of RSA, ElGamal, or elliptic curve cryptosystems or the Diffie-Hellman key exchange protocol [1]. For *modular reduction*, division is normally used, which can be performed via multiplication with the *reciprocal* of the divisor, so fast reciprocal calculation is also important. In most of these calculations, computing only parts of the full products are sufficient.

We present new speedup techniques for these and other arithmetic operations, critical for embedded applications. Optimization of memory usage was important, too, but it is not the subject of this paper.

For operand sizes of cryptographic applications (128,..., 4096 bits), school multiplication is used the most often (digit products summed up), requiring simple control structure. Speed improvements can be achieved with Karatsuba's method and the Toom-Cook 3- or 4-way multiplication, but asymptotically even faster algorithms are slower for these operand lengths [2, 3]. We consider digit-serial multiplication algorithms of time complexity $O(n^\alpha)$, $1 < \alpha \leq 2$, that is, no parallel- or discrete-Fourier-transform based techniques, which require different optimization methods (see [4]).

This paper is the second part of a manuscript accepted for CHES'05. Because of page limitations only the first half was presented and included in the proceedings [5]. All results of the full paper were introduced in the CHES'04 Rump session with the URL to the author's website, where the draft paper was available since early 2003.

The main results of [5], used in this paper, are the following.

(I) *Squaring is about twice faster than multiplication at $O(n^\alpha)$ complexity algorithms,* $1 < \alpha \leq 2$.

(II) *The LS and MS half products are of equal complexity, within an additive linear term.* (This is a nontrivial result, because the carry propagation has to be handled properly.)

(III) *The least speedup factors of specific truncated products are shown in Table 1.*

In tables below, new results are typeset in *italics*, and new formulas are boxed.

Table 1

Product	School	Karatsuba	Toom-Cook-3	Toom-Cook-4
Half: γ_α	0.5	0.8078	0.8881	0.9232
Middle third: δ_α	1	1	1.6434	1.6979
Third quarter	0.375	0.6026	0.9170	0.9907

2. TRUNCATED PRODUCTS

Truncated multiplication computes a *truncated product*, a contiguous subsequence of the digits of the product of two 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 school multiplication. For half products computed by Fourier-transform-type multiplications, no constant time speedup is known.

Fast truncated product algorithms are introduced and analyzed in [5]. A recursive procedure can be defined, when several smaller full or truncated products cover the desired digit sequence to be computed. In [5] such covers are investigated and the time complexity of the resulting algorithms are determined.

3. MODULAR ARITHMETIC IN CRYPTOGRAPHY

Messages and other types of data appear in computers as a sequence of bits, which can be interpreted as (long) integers. Encryption is to apply a (hard to invert) one-to-one transform on them. Such transforms can be constructed with common integer arithmetic operations, like additions and multiplications. To prevent (intermediate) results to grow too long, some measures are necessary. Binary truncation is not suitable in general, because the operation would not be invertible, which was useful at decryption. Modular arithmetic is better, with a fixed modulus, which is a huge prime number, or a product of two large primes in the commonly used cryptosystems.

Modular arithmetic is a system of arithmetic for integers, where numbers "wrap around" after they reach a certain value—the modulus, that is, larger numbers are replaced with their remainders of a division by the modulus. The operation of finding the remainder is the modulo operation, written as "mod": $10 \bmod 3 = 1$.

4. CRYPTOGRAPHIC APPLICATIONS

Symmetric-key cryptosystems typically use the same key for encryption and decryption. Its significant disadvantage is the key management involved. Each pair of communicating parties must share a different key. On the other hand, in public-key cryptosystems, the public key is freely distributed, while its paired private key is secret. The public key is typically used for encryption or signature verification, while the private or secret key is used for decryption or for digital signatures on documents.

Truncated products are most important in public-key cryptography, where long integer arithmetic is used, like at RSA, ElGamal, and elliptic curve cryptosystems, but there are many others. We will present speedup techniques for their basic operations after an overview of these cryptosystems. Details are in [1].

4.1. RSA cryptosystem

RSA encryption (decryption) of a message (ciphertext) g is done by modular exponentiation $g^e \bmod m$ with different encryption (e) and decryption (d) exponents, such that $(g^e)^d \bmod m = g$. The exponent e is the public key, together with the modulus $m = p \cdot q$, the product of two large primes. d is the corresponding private key. The security lies in the difficulty of factoring m.

4.2. ElGamal cryptosystem

The public key is (p, α, α^a), fixed before the encrypted communication, with randomly chosen α, a, and prime p. Encryption of the message m is done by choosing a random $k \in [1, p-2]$, computing $\gamma = \alpha^k \bmod p$ and $\delta = m \cdot (\alpha^a)^k \bmod p$.

Decryption is done with the private key a, by computing first the *modular inverse* of γ, then $(\gamma^{-1})^a = (\alpha^{-a})^k \bmod p$, and multiplying it to $\delta : \delta \cdot (\alpha^{-a})^k \bmod p = m$.

4.3. Elliptic curve cryptography

Prime field elliptic curve cryptosystems are gaining popularity especially in embedded systems, because of their smaller need in processing power and memory than RSA or ElGamal. An elliptic curve E over GF(p) (the field of residues modulo the prime $p > 2$) is defined as the set of points (x, y) (together with the point at infinity O) satisfying the reduced Weierstrass equation,

$$E : f(X, Y) \triangleq Y^2 - X^3 - a \cdot X - b \equiv 0 \bmod p. \quad (1)$$

The data to be encrypted is represented by a point P on a chosen curve. Encryption by the key k is performed by computing $Q = P+P+\cdots+P = k \cdot P$, called scalar multiplication (the additive notation for exponentiation). It is usually computed with variations of the *double-and-add* method. When the resulting point is not the point at infinity O, the addition of points $P = (x_P, y_P)$ *and* $Q = (x_Q, y_Q)$ leads to the resulting point R $= (x_R, y_R)$ through the following computation:

$$\begin{aligned} x_R &= \lambda^2 - x_P - x_Q \bmod p, \\ y_R &= \lambda \cdot (x_P - x_R) - y_P \bmod p, \end{aligned} \quad (2)$$

where

$$\begin{aligned} \lambda &= (y_P - y_Q) \cdot (x_P - x_Q)^{-1} \bmod p \quad \text{if } P = Q, \\ \lambda &= (3x_P^2 + a) \cdot (2y_P)^{-1} \bmod p \quad \text{if } P = Q. \end{aligned} \quad (3)$$

Table 2

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

The addition described above (and extended naturally to handle the point at infinity) is commutative and associative and defines an algebraic *group* on the points of the elliptic curve (with O being the neutral element, and the inverse of the point (x, y) being $(x, -y)$); see the details in [6].

5. TIME COMPLEXITY

Multiplication is more expensive (slower and/or more hardware consuming) even on single digits, than addition or store/load operations (or if single-cycle multiplications are implemented, they restrict the clock speed, like at ARM10). Many computing platforms perform additive- and data-movement operations parallel to multiplications (PowerPC, Pentium MMX, Athlon SSE, ARM10, most DSPs), so they do not 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 ($4, \ldots, 8$ machine words or more) the number of digit multiplications is well approximated by $n^{\alpha}(\approx M_{\alpha}(n))$, where α is listed in Table 2. (These are recursive algorithms, derived from polynomial interpolation, when the digits of the operands are treated as coefficients of the powers of the unknown. See, e.g., [7]. Here we only use them as black-box library functions.)

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 nonmultiplicative digit operations are executed, that cannot be performed 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. This is why there is *no need to measure running time* of the presented algorithms, in all different computing systems imaginable. Just use the often readily available speed ratios of the various full multiplication functions on that particular computing system.

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

Figure 1: 34.5-bit initial reciprocal.

6. RECIPROCAL

Reciprocals are used as building blocks of more complex modular arithmetic operations, like of Barrett's modular multiplication algorithm. They are often included in function libraries, which support cryptographic operations, protocols.

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 ($0.5 \le x < 1$) and scale (shift) the result after the reciprocal calculations to get the integer reciprocal $\mu = \lfloor d^{2n}/x \rfloor$.

The well-known Newton iteration is a 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 computed with a handful of operations. In 32-bit platforms 6-digit multiplications and 5 additions are enough, as shown in Figure 1, with constants in the innermost parentheses. The first line represents a linear approximation of the reciprocal of $0.5 \le x < 1$, followed by three slightly modified Newton iterations. The constants, resulted from numerical optimizations, provide the smallest worst case approximation error. On sufficiently precise arithmetic engines, it provides more than 34 bit accurate initial estimate r of $1/x$.

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

$$
\begin{aligned}
r &= \frac{1}{x}(1 - \varepsilon), \\
r \longleftarrow r \cdot (2 - rx) &= \frac{1}{x}(1 - \varepsilon)\big(2 - (1 - \varepsilon)\big) = \frac{1}{x}(1 - \varepsilon^2).
\end{aligned} \tag{4}
$$

If started with 32-bit accuracy, the iterations give approximate reciprocal values of $k = 64, 128, 256, \ldots$ 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 do not change, so we just calculate the necessary new digits and attach them to r:

$$
r_{k+1} = r_k \| \text{ digits } [2^k + 1, \ldots, 2^{k+1} \text{ of } r^2(-x)]. \tag{5}
$$

Table 3

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
0.5	0.9039	*1.4379*	*1.5927*

Table 4

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
0.625	1.1732	*1.7596*	*1.9416*

Having an $m = 2^k$-digit accurate reciprocal r_k, we perform an 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 $m+1 \cdots 2m$ have to be calculated. This is a *third-quarter* product [5]. With school multiplication it takes $1.5m^2$-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 $R_2(n) = 2(1 + 2^2 + 4^2 + \cdots + (n/2)^2) = 2/3n^2 - 2/3$. However, there is still a better way to organize the work.

Algorithm R. Arrange the calculation according to: $r_{k+1} \leftarrow r_k + r_k(1 - r_k x)$. Here, $r_k x \approx 1 - d^{-2k}$ (2^k-digit accuracy if we started with 1 accurate digit approximation), so the $m = 2^k$ MS digits of $r_k x$ 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 [5]). 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 $-x_{(2m)} := MS_{2m}(d^n - x)$):

$$r_{k+1} = r_k \| r_k \ltimes (r_k \otimes -x_{(2m)}). \tag{6}$$

The series of multiplications take

$$\boxed{(\delta_\alpha + \gamma_\alpha)\Sigma_{k=1,2,4\cdots n/2} M_\alpha(k)} \tag{7}$$

time. They sum up to the ratios shown in Table 3, compared to the corresponding multiplication time $M_\alpha(n)$.

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 conforms to our complexity requirements (fewer auxiliary operations than at multiplications).

We have left out all of the details with the rounding (see [8]). 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, or 90% of one Karatsuba multiplication.

Note 2. The speedup techniques in Algorithm R (concatenations instead of additions and the precalculation 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 [9], the results for the Toom-Cook multiplications are new.

7. 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 give the remainder. See more at Barrett's multiplication, below.) For cryptography the interesting case is $2n$-digit long dividend, over n-digit divisor. The quotient is also n-digit 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 [3] incorporates the final multiplication ($y \cdot 1/x$) into the last Newton iteration:

$$\begin{aligned} z_{n/2} &\longleftarrow r_{n/2} \ltimes y_{2n-1,\ldots,3n/2}, \\ \left\lfloor \frac{y}{x} \right\rfloor &= z_{n/2} \| r_{n/2} \ltimes (y_{3n/2-1,\ldots,n} - z_{n/2} \otimes x). \end{aligned} \tag{8}$$

The complexity of the final Newton iteration remains the same, but the multiplication step becomes faster: $\gamma_\alpha \cdot M_\alpha(n/2) = \gamma_\alpha \cdot M_\alpha(n)/2^\alpha$. It is added to the complexity expression of the reciprocal above, giving the complexity of calculating the quotient of a $2n$-digit dividend over an n-digit divisor (relative to $M_\alpha(n)$) (see Table 4). 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 speed is closer to $1.3 \cdot M_\alpha(n)$: [8]), 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 (RSA elliptic curves) many divisions are performed with the same divisor (the fixed modulus at modular reduction). In this case, the time for calculating the reciprocal becomes negligible compared to the time consumed by the large number of multiplications, so the amortized cost of the division is only one *half multiplication:* $\boxed{\gamma_\alpha \cdot M_\alpha(n)}$.

Historical note

In [9] the Karatsuba case was analyzed and also a faster direct division algorithm was presented, which reduces the coefficient for the Karatsuba division to 1 (14.5% speedup). Unfortunately, the direct division algorithm needs complicated carry-save techniques, which increase the number of auxiliary operations well beyond the limit of our complexity model. In [10] practical direct division with Karatsuba

$(A_1 d^n + A_0) \longleftarrow a \cdot b$
$q \longleftarrow A_1 \ltimes \mu$
$r \longleftarrow A_0 - q \rtimes m$
if $r < 0 : r \longleftarrow r + d^{n+1}$
while $r \geq m : r \longleftarrow r - m$

Figure 2: Barrett's multiplication.

complexity was presented. Its empirical complexity coefficient was around 2 on a particular computer, but that includes all the nonmultiplicative operations, so we cannot directly compare it to the results here.

8. BARRETT MULTIPLICATION

Modular multiplications can start with regular multiplications, then we subtract the modulus as many times as possible ($q = \lfloor ab/m \rfloor$), keeping the result nonnegative. The most significant half of the product is enough to determine this quotient, which is computed fast by multiplication with the precomputed reciprocal of m. Using only the necessary digits, which influence the result, leads to the following.

Algorithm B.

$$ab \bmod m = ab - \left\lfloor \frac{ab}{m} \right\rfloor m = \mathrm{LS}(ab) - (\mathrm{MS}(ab) \ltimes \mu) \rtimes m \quad \text{with } \mu = \left\lfloor \frac{d^{2n}}{m} \right\rfloor. \tag{9}$$

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-digit 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 are assumed to be calculated faster than the full products.

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 [11]. Also, μ and q require $2n$-digit 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 precomputation is simple and fast (taking less time than one modular multiplication in case of school, or Karatsuba multiplication).

The dominant part of the time complexity of Barrett's multiplication is $\boxed{(1 + 2\gamma_\alpha)M_\alpha(n)}$, a significant improvement over the previous best results of 3 (2 for the school multiplication) in [10]. The speed ratios over $M_\alpha(n)$ are shown in Table 5.

Table 5

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
2	*2.6155*	*2.7762*	*2.8464*

Table 6

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
1.5	*2.1155*	*2.2762*	*2.3464*

Table 7

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
1.5	*1.8078*	*2.5316*	*2.6211*

Modular squaring

Since the nonmodular square a^2 is calculated twice faster than the general products ab [5], the first step of the Barrett multiplication becomes faster. The rest of the algorithm is unchanged, giving the speed ratio $\boxed{0.5 + 2\gamma_\alpha}$ of modular squaring over the $n \times n$-digit multiplication time $M_\alpha(n)$ (see Table 6).

Constant operand

With precalculations 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 precalculate the n digits long $\beta' := MS_n(b/m)$. With it

$$a \cdot b \bmod m = a \rtimes b - (a \ltimes \beta') \rtimes m. \tag{10}$$

The corresponding algorithm runs faster, in $3\gamma_\alpha M_\alpha(n) < (1 + 2\gamma_\alpha)M_\alpha(n)$ time. Another idea is expressing the modular multiplication, with fractional parts: $ab \bmod m = \{ab/m\}m$. This leads to an even faster algorithm.

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

$$a \cdot b \bmod m = (a \otimes \beta) \ltimes m. \tag{11}$$

For Barrett multiplications with constants, this equation gives speed ratios over $M_\alpha(n)$: $\boxed{(\delta_\alpha + \gamma_\alpha)M_\alpha(n)}$. It is close to the squaring time, in the Karatsuba case even faster representing a significant *improvement* over previously used general multiplication algorithms (see Table 7).

Note 1. Barrett's modular multiplication calculates the quotient $q = \lfloor ab/m \rfloor$ as well (line 2 in Figure 2). If it is needed

```
for i = 0 · · · n − 1
    t = x_i · m′ mod d
    x = x + t · m · d^i
x = x/d^n
if (x ≥ m)
    x = x − m
```

Figure 3: Montgomery reduction.

```
x_0 = 0
for  i = 0 · · · n − 1
     t = (x_0 + a_i · b_0) · m′ mod d
     x = (x + a_i · b + t · m)/d
if (x ≥ m)
   x = x − m
```

Figure 4: Montgomery multiplication (a_i, b_i = ith digit of a, b).

outside of the function, the final correction step (while-loop) has to increment its value, too.

Note 2. As originally published in [12], Barrett's modular multiplication was the first example of the use of truncated products in cryptographic algorithms, but no subquadratic version was presented there.

9. MONTGOMERY MULTIPLICATION

As originally formulated in [13] 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 [14]). 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 straightforward implementation of the direct division based reduction, using school multiplications [11]. More sophisticated implementations of the direct division based method actually outperform Montgomery's original multiplication algorithm [15].

Montgomery's multiplication calculates the product in "row order," so a little tweaking is necessary for speeding it up at squaring [15]. 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, that is, pre- and post-processing is necessary, each taking time comparable to one modular multiplication.

In Figure 3, the Montgomery reduction is described. The digits of x are denoted by $x_{n-1}, \ldots, x_1, x_0$. The digits on the left of the currently processed one are constantly updated. The single digit $m' = -m_0^{-1} \bmod d$ is a pre-calculated constant, which exists if m_0 (the least significant digit of m) and d are relative primes. In cryptography, m_0 is odd, because m is a large prime or a product of large primes; and d is a power of two, being the base of the number system using computer words as digits.

The rationale behind the algorithm is representing a long integer $a, 0 \le a < m$, as $a \cdot \mathrm{R} \bmod m$ with $\mathrm{R} = d^n$. The modular product of two numbers in this representation is $(a\mathrm{R})(b\mathrm{R}) \bmod m$, which is converted to the right form by multiplying with R^{-1}, since $(a\mathrm{R})(b\mathrm{R})\mathrm{R}^{-1} \bmod m = (ab)\mathrm{R} \bmod m$. This correction step, $x \to x \cdot \mathrm{R}^{-1} \bmod m$ 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).

Table 8

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
2	*2.6155*	*2.7762*	*2.8464*

Table 9

School	Karatsuba	Toom-Cook-3	Toom-Cook-4
1.5	*2.4233*	*2.6644*	*2.7696*

9.1. Montgomery multiplications with truncated products

Montgomery's reduction implicitly finds u (the t values constitute 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 $\bmod d^n : -x \equiv u \cdot m \bmod d^n$, or $u = x \cdot (-m^{-1}) \bmod d^n = x \rtimes (-m^{-1})$. Here $-m^{-1} \bmod d^n$ can be pre-calculated with any modular inverse algorithm (m is odd, $d = 2^k$). Again, using only those digits, which influence the result leads to the following Montgomery reduction.

Algorithm M.

$$x \cdot d^{-n} \bmod m = \mathrm{MS}(x) - \left(\mathrm{LS}(x) \rtimes \left(-m^{-1}\right)\right) \ltimes m. \tag{12}$$

With two half products and one full multiplication (to get $x = a \cdot b$) the above algorithm takes exactly as much time as the Barrett multiplication $\boxed{(1 + 2\gamma_\alpha)M_\alpha(n)}$ (with the same squaring speedup possibility as at Barrett multiplication) (see Table 8).

Algorithm MC. Montgomery multiplication with constants is calculated in $\boxed{3\gamma_\alpha \cdot M_\alpha(n)}$ time with

$$\begin{aligned} \beta &:= b \rtimes \left(-m^{-1}\right), \\ abd^{-n} \bmod m &= a \ltimes b - (a \rtimes \beta) \ltimes m \end{aligned} \tag{13}$$

(see Table 9).

Note

These are new, faster algorithms for the Montgomery multiplication using Karatsuba or Toom-Cook multiplications. Employing the squaring and precomputed constant variants even of the quadratic time algorithms, long exponentiations can be performed faster than with the original Montgomery multiplications. However, the advantage of the simplicity of the right-to-left division is gone, but the costs of pre- and post processing remain. Therefore, there is no speed advantage of using Montgomery multiplications until further significant accelerations are found (but the original code or hardware can be simpler). Currently, Barrett's method is faster. If storage space is a concern, direct division-based modular reductions work better, not needing precomputation or extra memory [9, 10, 15].

In [16] there is a hint (but no reference) that a commercial application might use Montgomery multiplication of Karatsuba complexity. It appeared more than two years after our results above were first made public, and their algorithm is not described.

10. 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 designs 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 [17] modified in [10], 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 used).

The coprocessor has to perform $n/2$-by-$n/2$-digit full 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.

10.1. Algorithm Q1

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

$$\begin{aligned} a &= (a_1, a_0) = a_1 d^n + a_0, \\ b &= (b_1, b_0) = b_1 d^n + b_0, \\ m &= (m_1, m_0) = m_1 d^n + m_0, \\ 0 &\le a_1, a_0, b_1, b_0, m_1, m_0 < d^n. \end{aligned} \tag{14}$$

We assume that m is normalized, that is, $d^{2n}/2 \le m < d^{2n}$. 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:

$$\begin{aligned} L &= \mathrm{LS}(a_0 b_1) + \mathrm{LS}(a_1 b_0), \\ M &= \mathrm{MS}(a_0 b_1) + \mathrm{MS}(a_1 b_0). \end{aligned} \tag{15}$$

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.

(i) The largest term $a_1 b_1 d^{2n}$ is reduced by n-digits with modular multiplication $(q_1, r_1) \leftarrow \mathrm{ModMult}(a_1, b_1, m_1)$. Taking $d^n q_1 m$ from ab cancels the MS n digits.

(ii) $d^n(d^n(r_1 + M) + L - q_1 m_0) + a_0 b_0$ is left. Cancel the MS digit as before with modular reduction:

$$(q_2, r_2) \longleftarrow \mathrm{ModRed}(d^n(r_1 + M) + L - q_1 m_0, m_1). \tag{16}$$

Note that the first argument of ModRed is $2n$-digit 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 $q_2 m$. It leaves

$$R = d^n(r_2 + L) - q_2 m_0 + a_0 b_0. \tag{17}$$

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 $0 \le R < m$.

Proposition 1. *Algorithm Q1 computes the $2n$-digit $(ab \bmod m)$ with at most 16 half multiplications of n-digits and one (precomputed) n-digit reciprocal.*

Proof. $\mathrm{R} = ab - k \cdot m$ for some integer k, and $0 \le \mathrm{R} < m$ $\mathrm{R} = ab \bmod m$.

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. □

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.

10.2. 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,

$(q_1, r_1) = \text{ModMult}\,(a_1, b_1, m_1)$
$M = a_0 \ltimes b_1 + a_1 \ltimes \mathbf{b}_0$
$x_1 = (q_1 \ltimes m_0, q_1 \rtimes m_0)$
$(q_2, r_2) = \text{ModRed}\,(d^n(r_1 + M) - x_1, m_1)$
$L = a_0 \rtimes b_1 + a_1 \rtimes b_0$
$x_2 = (q_2 \ltimes m_0, q_2 \rtimes m_0)$
$c_0 = (a_0 \ltimes b_0, a_0 \rtimes b_0)$
$R = d^n(r_2 + L) - x_2 + c_0$
while $(R \geq m)$
    $R = R - m$
while $(R < 0)$
    $R = R + m$

Figure 5: Q1 quad-length modular multiplication.

$c_{11} = a_1 \ltimes b_1$; $c_{10} = a_1 \rtimes b_1$
$c_{01} = a_0 \ltimes b_0$; $c_{00} = a_0 \rtimes b_0$
$M = c_{01} + c_{11} + (a_1 - a_0) \ltimes (b_1 - b_0)$
$L = c_{00} + c_{10} + (a_1 - a_0) \rtimes (b_1 - b_0)$
$(q_1, r_1) = \text{ModRed}\,(d^n c_{11} + c_{10}, m_1)$
$x_1 = (q_1 \ltimes m_0, q_1 \rtimes m_0)$
$(q_2, r_2) = \text{ModRed}\,(d^n(r_1 + M) - x_1, m_1)$
$x_2 = (q_2 \ltimes m_0, q_2 \rtimes m_0)$
$R = d^n(r_2 + L) - x_2 + c_0$
while $(R \geq m)$
    $R = R - m$
while $(R < 0)$
    $R = R + m$

Figure 6: Q2 quad-length modular multiplication.

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

$$L = \text{LS}\,(a_0 b_0) + \text{LS}\,(a_1 b_1) - \text{LS}\,((a_1 - a_0)(b_1 - b_0)),$$
$$M = \text{MS}\,(a_0 b_0) + \text{MS}\,(a_1 b_1) - \text{MS}\,((a_1 - a_0)(b_1 - b_0)). \quad (18)$$

The product $a \cdot b$ can still 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, 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 $c_1 = a_1 b_1$ has already been calculated.

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

Proof. $\mathrm{R} = ab - k \cdot m$ for some integer k, and $0 \leq \mathrm{R} < m$ $\mathrm{R} = ab \bmod m$. 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 μ_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.

11. 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 are as follows.

(i) Fast 32-bit initialization of the Newton reciprocal algorithm.
(ii) Fast Newton's reciprocal algorithm with only truncated product arithmetic (without any external additions or subtractions).
(iii) New long integer division algorithms based on Toom-Cook multiplications.
(iv) Accelerated Barrett multiplication with Karatsuba complexity and faster.
(v) Speedup of Barrett's squaring and multiplication when one multiplicand is constant.
(vi) New algorithms for Montgomery multiplication with Karatsuba complexity and faster.
(vii) Speedup of Montgomery squaring and multiplication when one multiplicand is constant, with sub-quadratic and also with quadratic complexity.
(viii) Fast and adaptable quad-length modular multiplications on short arithmetic coprocessors.

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 hardware or software features constraints. (Our implementation and simulation results are being collected in a separate paper [8].) The results are very much dependent on the characteristics of the hardware (word length; special instructions; parallel instructions; instruction timings; instruction pipeline; cache; memory size and layout; virtual/paging memory...) and the used software (operating system; compiler; concurrent tasks; active processes...). This is why it is important that the speed of the presented algorithms is roughly proportional to the speed of the underlying multiplication algorithms, that is, knowing the fastest full multiplication algorithm for a specific computing platform tells, which version of the presented algorithms is the fastest (within a small margin).

Notations

(1) Long integers are denoted by $A = (a_{n-1} \cdots a_1 a_0) = a_{n-1 \cdots 0} = \Sigma d^i a_i$ in a d-ary number system, where a_i, $0 \le a_i \le d-1$ are the *digits* (usually 16 or 32 bits: $d = 2^{16}$ or 2^{32})
(2) $|A|$ denotes the number of digits, the length of a d-ary number $|(a_{n-1} \cdots a_1 a_0)| = n$.
(3) $A \| B$ is the number of the joined digit-sequence $(a_{n-1} \cdots a_0 b_{m-1} \cdots b_0)$; $|A| = n$, $|B| = m$.
(4) $\lfloor x \rfloor$ denotes the integer part (floor) of x, and $0 \le \{x\} < 1$ is the fractional part, such that $x = \lfloor x \rfloor + \{x\}$.
(5) $\log n = \log_2 n = \log n / \log 2$.
(6) LS stands for *l*east *s*ignificant, the low-order bit/s or digit/s of a number.
(7) MS stands for *m*ost *s*ignificant, the high-order bit/s or digit/s of a number.
(8) (*grammar*) *School multiplication denotes division*: the digit-by-digit multiplication and division algorithms, as taught in elementary schools.
(9) $A \ltimes B$, $A \rtimes B$ denote the MS or LS half of the digit-sequence of the product $A \cdot B$, respectively.
(10) $A \otimes B$ denotes the middle third of the digit sequence of $A \cdot B$.
(11) $M_\alpha(n)$ is the time complexity of a Toom-Cook-type full multiplication algorithm, $M_\alpha(n) = O(n^\alpha)$, with $1 < \alpha \le 2$.
(12) γ_α = the speedup factor of the half multiplication, relative to $M_\alpha(n)$.
(13) δ_α = the speedup factor of the middle-third product, relative to $M_\alpha(n)$.

REFERENCES

[1] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, Fla, USA, 1996.

[2] *GNU multiple precision arithmetic library manual*, http://www.swox.com/gmp/.

[3] A. H. Karp and P. Markstein, "High-precision division and square root," *ACM Transactions on Mathematical Software*, vol. 23, no. 4, pp. 561–589, 1997.

[4] D. J. Bernstein, "Fast Multiplication and its Applications," http://cr.yp.to/papers.html#multapps.

[5] L. Hors, "Fast truncated multiplication for cryptographic applications," in *Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '05)*, vol. 3659 of *Lecture Notes in Computer Science*, pp. 211–225, Edinburgh, UK, August 2005.

[6] N. Koblitz, *Introduction to Elliptic Curves and Modular Forms*, Springer, New York, NY, USA, 1984.

[7] D. E. Knuth, *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading, Mass, USA, 1981.

[8] L. Hars, "Multiplications for Cryptographic Operand Lengths: Analytic and Experimental Comparisons," manuscript.

[9] G. Hanrot, M. Quercia, and P. Zimmermann, "The middle product algorithm, I," Rapport de Recherche 4664, l'Institut National de Recherche en Informatique et en Automatique, Lorraine, France, 2002, http://www.inria.fr/rrrt/rr-4664.html.

[10] C. Burnikel and J. Ziegler, "Fast recursive division," MPI Research Report I-98-1-022, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1998.

[11] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of three modular reduction functions," in *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '93)*, vol. 773 of *Lecture Notes in Computer Science*, pp. 175–186, Santa Barbara, Calif, USA, August 1994.

[12] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Proceedings of International Cryptology Conference on Advances in Cryptology (CRYPTO '86)*, pp. 311–323, Santa Barbara, Calif, USA, 1987.

[13] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[14] K. Hensel, *Theorie der Algebraischen Zahlen*, Teubner, Leipzig, Germany, 1908.

[15] L. Hars, "Long modular multiplication for cryptographic applications," in *Proceedings of the 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '04)*, pp. 45–61, Cambridge, Mass, USA, August 2004.

[16] Shamus Software Ltd, *MIRACL users manual*, version 5.0, December 2005, ftp://ftp.computing.dcu.ie/pub/crypto/manual.zip.

[17] W. Fischer and J.-P. Seifert, "Increasing the bitlength of a crypto-coprocessor via smart hardware/software co-design," in *Proceedings of 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '02)*, vol. 2523 of *Lecture Notes in Computer Science*, pp. 71–81, Redwood Shores, Calif, USA, August 2002.