Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2006, Article ID 32192, Pages 1–13
DOI 10.1155/ES/2006/32192

Modular Inverse Algorithms Without Multiplications for Cryptographic Applications

Laszlo Hars

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

Received 19 July 2005; Revised 1 December 2005; Accepted 17 January 2006

Recommended for Publication by Sandro Bartolini

Hardware and algorithmic optimization techniques are presented to the left-shift, right-shift, and the traditional Euclidean-modular inverse algorithms. Theoretical arguments and extensive simulations determined the resulting expected running time. On many computational platforms these turn out to be the fastest known algorithms for moderate operand lengths. They are based on variants of Euclidean-type extended GCD algorithms. On the considered computational platforms for operand lengths used in cryptography, the fastest presented modular inverse algorithms need about *twice* the time of modular multiplications, or even less. Consequently, in elliptic curve cryptography delaying modular divisions is slower (affine coordinates are the best) and the RSA and ElGamal cryptosystems can be accelerated.

1. INTRODUCTION

We present improved algorithms for computing the inverse of large integers modulo a given prime or composite number, without multiplications of any kind. In most computational platforms they are much faster than the commonly used algorithms employing multiplications, therefore, the multiplier engines should be used for other tasks in parallel. The considered algorithms are based on different variants of the Euclidean-type greatest common divisor algorithms. They are iterative, gradually decreasing the length of the operands and keeping some factors updated, maintaining a corresponding invariant. There are other algorithmic approaches, too. One can use system of equations or the little Fermat theorem (see [1]), but they are only competitive with the Euclidean-type algorithms under rare, special circumstances.

Several variants of three extended GCD algorithms are modified for computing modular inverses for operand lengths used in public key cryptography (128 bits–16 Kb). We discuss algorithmic improvements and simple hardware enhancements for speedups in digit-serial hardware architectures. The main point of the paper is to investigate *how much improvement* can be expected from these optimizations. It helps implementers to choose the fastest or smallest algorithm; allows system designer to estimate accurately the response time of security systems; facilitates the selection of the proper point representation for elliptic curves, and so forth.

The discussed algorithms run in quadratic time: $O(n^2)$ for n-bit input. For very long operands more complex algorithms such as Schönhage's half-GCD algorithm [2] get faster, running in $O(n \log^2 n)$ time, but for operand lengths used in cryptography they are far too slow (see [3]).

1.1. *Extended greatest common divisor algorithms*

Given 2 integers x and y the *extended* GCD algorithms compute their greatest common divisor g, and also two integer *factors* c and d: $[g, c, d] = x\text{CGD}(x, y)$, such that $g = c \cdot x + d \cdot y$. For example, the greatest common divisor of 6 and 9 is 3; and $3 = (-1) \cdot 6 + 1 \cdot 9$.

In the sequel we will discuss several xGCD algorithms. (See also [4] or [5].) They are iterative, that is, their input parameters get gradually decreased, while keeping the GCD of the parameters unchanged (or keep track of its change). The following relations are used:

(i) $\text{GCD}(x, y) = \text{GCD}(x \pm y, y)$,
(ii) $\text{GCD}(x, y) = 2 \cdot \text{GCD}(x/2, y/2)$ for even x and even y,
(iii) $\text{GCD}(x, y) = \text{GCD}(x/2, y)$ for even x and odd y.

1.2. *Modular inverse*

The positive residues $1, 2, \ldots, p - 1$ of integers modulo p (a prime number) form a multiplicative group G, that is, they obey the following 4 group laws.

(1) Closure: if x and y are two elements in G, then the product $x \cdot y := xy \bmod p$ is also in G.
(2) Associativity: the defined multiplication is associative, that is, for all $x, y, z \in \mathrm{G} : (x \cdot y) \cdot z = x \cdot (y \cdot z)$.
(3) Identity: there is an identity element $i (= 1)$ such that $i \cdot x = x \cdot i = x$ for every element $x \in \mathrm{G}$.
(4) Inverse: there is an inverse (or reciprocal) x^{-1} of each element $x \in \mathrm{G}$, such that $x \cdot x^{-1} = i$.

The inverse mentioned in (4) above is called the *modular inverse*, if the group is formed by the positive residues modulo a prime number. For example the inverse of 2 is $3 \bmod 5$, because $2 \cdot 3 = 6 = 1 \bmod 5$.

Positive residues modulo a composite number m do not form a group, as some elements do not have inverse. For example, 2 has no inverse mod 6, because every multiple of 2 is even, never $1 \bmod 6$. Others, like 5 do have inverse, also called modular inverse. In this case the modular inverse of 5, $5^{-1} \bmod 6$, is also 5, because $5 \cdot 5 = 25 = 24 + 1 = 1 \bmod 6$. In general, if x is relative prime to m (they share no divisors), there is a modular inverse $x^{-1} \bmod m$. (See also in [4].)

Modular inverses can be calculated with any of the numerous xGCD algorithms. If we set $y = m$, by knowing that $\mathrm{GCD}(x, m) = 1$, we get $1 = c \cdot x + d \cdot m$ from the results of the xGCD algorithm. Taking this equation modulo m we get $1 = c \cdot x$. The modular inverse is the smallest positive such c, so either $x^{-1} = c$ or $x^{-1} = c + m$.

1.3. Computing the xGCD factors from the modular inverse

In embedded applications the code size is often critical, so if an application requires both xGCD and modular inverse, usually xGCD is implemented alone, because it can provide the modular inverse, as well. We show here that from the modular inverse the two xGCD factors can be reconstructed, even faster than it would take to compute them directly. Therefore, it is always *better to implement a modular inverse* algorithm than xGCD. These apply to subroutine libraries, too, there is no need for a full xGCD implementation.

The modular inverse algorithms return a positive result, while the xGCD factors can be negative. $c = x^{-1}$ and $c = x^{-1} - y$ provide the two minimal values of one xGCD factor. The other factor is $d = (1 - c \cdot x)/y$, so $d = (1 - x \cdot x^{-1})/y$ and $d = x + (1 - x \cdot x^{-1})/y$ are the two minimal values. One of the c values is positive, the other is negative, likewise d. We pair the positive c with the negative d and vice versa to get the two sets of minimal factors.

To get d, calculating only the MS half of $x \cdot x^{-1}$, plus a couple of guard digits, is sufficient. Division with y provides an approximate quotient, which rounded to the nearest integer gives d. This way there is no need for longer than $\|y\|$-bit arithmetic (except two extra digits for the proper rounding). The division is essentially of the same complexity as multiplication (for operand lengths in cryptography it takes between 0.65 and 1.2 times as long, see, e.g., [6]).

For the general case $g > 1$ we need a trivial modification of the modular inverse algorithms: return the last candidate for the inverse before one of the parameters becomes 0 (as noted in [7] for polynomials). It gives x^* such that $x \cdot x^* \equiv g \bmod y$. Again $c = x^*$ or $c = x^* - y$ and $d = (g - x \cdot x^*)/y$ or $d = x + (g - x \cdot x^*)/y$.

The extended GCD algorithm needs storage room for the 2 factors in addition to its internal variables. They get constantly updated during the course of the algorithm. As described above, one can compute the factors from the modular inverse and save the memory for one (long integer) factor and all of the algorithmic steps updating it. The xGCD algorithms applied for operand lengths in cryptography perform a number of iterations proportional to the length of the input, and so the operations on the omitted factor would add up to at least as much work as a shift-add multiplication algorithm would take. With a better multiplication (or division) algorithm not only memory, but also some computational work can be saved.

1.4. Cryptographic applications

The modular inverse of long integers is used extensively in cryptography, like for RSA and ElGamal public key cryptosystems, but most importantly in elliptic curve cryptography.

1.4.1. RSA

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 2 large primes. d is the corresponding private key. The security lies in the difficulty of factoring m. (See [5].) Modular inverse is used in the following.

(i) Modulus selection: in primality tests (excluding small prime divisors). If a random number has no modular inverse with respect to the product of many small primes, it proves that the random number is not prime. (In this case a simplified modular inverse algorithm suffice, which only checks if the inverse exists.)
(ii) Private key generation: computing the inverse of the chosen public key (similar to the signing/verification keys: the computation of the private signing key from the chosen public signature verification key). $d = e^{-1} \bmod (p-1)(q-1)$.
(iii) Preparation for CRT (Chinese remainder theorem based computational speedup): the precalculated half-size constant $C_2 = p^{-1} \bmod q$ (where the public modulus $m = p \cdot q$) helps accelerating the modular exponentiation about 4-fold [5].
(iv) Signed bit exponent recoding: expressing the exponent with positive and negative bits facilitates the reduction of the number of nonzero signed bits. This way many multiplications can be saved in the multiply-square binary exponentiation algorithm. At negative exponent bits the inverse of the message $g^{-1} \bmod m$—which almost always exists and precomputed in less time than 2 modular multiplications—is multiplied to

the partial result [8]. (In embedded systems, like smart cards or security tokens RAM is expensive, so other exponentiations methods, like windowing, are often inapplicable.)

1.4.2. ElGamal encryption

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 $k \in [1, p-2]$ and 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 $\delta : \delta \cdot (\alpha^{-a})^k \bmod p = m$. (See also in [5].)

1.4.3. Elliptic curve cryptography

Prime field elliptic curve cryptosystems (ECC) are gaining popularity especially in embedded systems, because of their smaller need in processing power and memory than RSA or ElGamal. Modular inverses are used extensively during point addition, doubling and multiplication (see more details in [4]). 20–30% overall speedup is possible, just with the use of a better algorithm.

An elliptic curve E over GF(p) (the field of residues modulo the prime p) is defined as the set of points (x, y) (together with the point at infinity O) satisfying the reduced Weierstraß equation:

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

In elliptic curve cryptosystems the data to be encrypted is represented by a point P on a chosen curve. Encryption by the key k is performed by computing $Q = P + P + \cdots + P = k \cdot P$. Its security is based on the hardness of computing the discrete logarithm in groups. This operation, called scalar multiplication (the additive notation for exponentiation), is usually computed with the *double-and-add* method (the adaptation of the well-known *square-and-multiply* algorithm to elliptic curves, usually with signed digit recoding of the exponent [8]). 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 $\mathrm{R} = (x_\mathrm{R}, y_\mathrm{R})$ through the following computation:

$$\begin{aligned} x_\mathrm{R} &= \lambda^2 - x_P - x_Q \bmod p, \\ y_\mathrm{R} &= \lambda \cdot (x_P - x_\mathrm{R}) - y_P \bmod p, \end{aligned} \quad (2)$$

where

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

Here the divisions in the equations for λ are shorthand notations for multiplications with the *modular inverse* of the denominator. $P = (x_P, y_P)$ is called the affine representation of the elliptic curve point, but it is also possible to represent points in other coordinate systems, where the field divisions (multiplications with modular inverses) are traded to a larger number of field additions and multiplications. These other point representations are advantageous when computing the modular inverse is much slower than a modular multiplication. In [9] the reader can find discussions about point representations and the corresponding costs of elliptic curve operations.

2. HARDWARE PLATFORMS

2.1. Multiplications

There are situations where the modular inverse has to be or it is better calculated without any multiplication operations. These include

(i) if the available multiplier hardware is slow,
(ii) if there is no multiplier circuit in the hardware at all. For example, on computational platforms where long parallel adders perform multiplications by repeated shift-add operations, (see [10] for fast adder architectures.)
(iii) for RSA key generation in cryptographic processors, where the multiplier circuit is used in the background for the exponentiations of the (Miller-Rabin) primality test [5],
(iv) in prime field elliptic or hyper elliptic curve cryptosystems, where the inversion can be performed parallel to other calculations involving multiplications.

Of course, there are also computational platforms, where multiplications are better used for modular inverse calculations. These include workstations with very fast or multiple multiplier engines (could be three: ALU, floating point multiplier, and multimedia extension module).

In *digit-serial* arithmetic engines there is usually a digit-by-digit multiplier circuit (for 8–128 bit operands), which can be utilized for calculating modular inverses. This multiplier is the slowest circuit component; other parts of the circuit can operate at much higher clock frequency. Appropriate hardware designs, with faster non-multiplicative operations, can defeat the speed advantage of those modular inverse algorithms, which use multiplications. This way faster and less expensive hardware cores can be designed.

This kind of hardware architecture is present in many modern microprocessors, like the Intel Pentium Processors. They have 1 clock cycle base time for a 32 bit integer add or subtract instruction (discounting operand fetch and other overhead), and they can sometimes be paired with other instructions for concurrent execution. A 32 bit multiply takes 10 cycles (a divide takes 41 cycles), and neither can be paired.

2.2. Shift and memory fetch

The algorithms considered in this paper process the bits or digits of their long operands sequentially, so in a single cycle fetching more neighboring digits (words) into fast registers allows the use of slower, cheaper RAM, or pipeline registers.

We will use only add/subtract, compare and shift operations. With trivial hardware enhancements the shift operations can be done "on the fly" when the operands are loaded for additions or subtractions. This kind of parallelism is customarily provided by DSP chips, and it results in a close to two-fold speedup of the shifting xGCD-based modular inverse algorithms.

Shift operations could be implemented with manipulating pointers to the bits of a number. At a subsequent addition/subtraction the hardware can provide the parameter with the corresponding offset, so arbitrary long shifts take only a constant number of operations with this offset-load hardware support. (See [11].) Even in traditional computers these pointer manipulating shift operations save time, allowing multiple shift operations to be combined into a longer one.

2.3. Number representation

For multidigit integers signed magnitude number representation is beneficial. The binary length of the result is also calculated at each operation (without significant extra cost), and pointers show the position of the most and least significant bits in memory.

(i) *Addition* is done from right to left (from the least to the most significant bits), the usual way.

(ii) *Subtraction* needs a scan of the operand bits from left to right, to find the first different pair. They tell the sign of the result. The leading equal bits need not be processed again, and the right-to-left subtraction from the *larger* number leaves no final borrow. This way subtraction is of the same speed as addition, like with 2's complement arithmetic.

(iii) *Comparisons* can be done by scanning the bits from left to right, too. For uniform random inputs the expected number of bit operations is constant, less than $1 \cdot 1/2 + 2 \cdot 1/4 + 3 \cdot 1/8 \ldots = 2$.

(iv) *Comparisons to* 0, 1, *or* 2^k take constant time also in the *worst case*, if the head and tail pointers have been kept updated.

3. MODULAR INVERSE ALGORITHMS

We consider all three Euclidean-type algorithm families commonly used: the extended versions of the right-shift, the left-shift, and the traditional Euclidean-algorithm. They all gradually reduce the length of their operands in an iteration, maintaining some invariants, which are closely related to the modular inverse.

3.1. Binary right shift: algorithms RS

At the modular inverse algorithm based on the *right-shift binary extended GCD* (variants of the algorithm of Penk, see in [12, Exercise 4.5.2.39] and [13]), the modulus m must be odd. The trailing 0 bits from two internal variables U and V

```
U ← $m$; V ← $a$;
R ← 0; S ← 1;
while (V > 0) {
      if ($U_0$ = 0) {
          U ← U/2;
          if ($R_0$ = 0)  R ← R/2;
          else R ← (R + $m$)/2;
      }
      else if ($V_0$ = 0) {
          V ← V/2;
          if ($S_0$ = 0) S ← S/2;
          else S ← (S + $m$)/2;
      }
      else //  U, V odd
          if (U > V) {
              U ← U − V; R ← R − S;
/ ∗ ∗/        if (R < 0) R ← R + $m$; }
          else {
              V ← V − U; S ← S − R;
/ ∗ ∗/        if (S < 0) S ← S + $m$; }
}
if (U > 1) return 0;
if (R > $m$) R ← R − $m$;
if (R < 0) R ← R + $m$;
return R; //  $a^{-1}$ mod $m$
```

Algorithm 1: Right-shift binary algorithm.

(initialized to the input a, m) are removed by shifting them to the right, then their difference replaces the larger of them. It is even, so shifting right removes the new trailing 0 bits (Algorithm 1).

Repeat these until V = 0, when U = GCD(m, a). If U > 1, there is no inverse, so we return 0, which is not an inverse of anything.

In the course of the algorithm two auxiliary variables, R and S, are kept updated. At termination R is the modular inverse.

3.1.1. Modification: algorithm RS1

The two instructions marked with "/ ∗ ∗/" in Algorithm 1. keep R and S nonnegative and so assure that they do not grow too large (the subsequent subtraction steps decrease the larger absolute value). These instructions are slow and not necessary, if we ensure otherwise, that the intermediate values of R and S do not get too large.

Handling negative values and fixing the final result is easy, so it is advantageous if instead of the marked instructions, we only check at the add-halving steps ($R \leftarrow (R+m)/2$ and $S \leftarrow (S+m)/2$) whether R or S was already larger (or longer) than m, and *add or subtract m* such that the result becomes smaller (shorter). These steps cost no additional work beyond choosing "+" or "−" and, if $|R| \le 2m$ was beforehand, we get $|R| \le m$, the same as at the simple halving of $R \leftarrow R/2$ and $S \leftarrow S/2$. If $|R| \le m$ and $|S| \le m$, $|R - S| \le 2m$ (the length could increase by one bit) but these instructions are always followed by halving steps, which prevent R and

S to grow larger than $2m$ during the calculations. (See code details at the plus-minus algorithm below.)

3.1.2. Even modulus

This algorithm cannot be used for RSA key generation, because m must be odd (to ensure that either R or R $\pm m$ is even for the subsequent halving step). We can go around the problem by swapping the role of m and a (a must be odd, if m is even, otherwise there is no inverse). The algorithm returns $m^{-1} \bmod a$, such that $m \cdot m^{-1} + k' \cdot a = 1$, for some negative integer $k' \cdot k' \equiv a^{-1} \bmod m$, easily seen if we take both sides of the equation mod m. It is simple to compute the smallest positive $k \equiv k' \bmod m$:

$$k = a^{-1} \bmod m = m + (1 - m \cdot m^{-1})/a. \qquad (4)$$

As we saw before, the division is fast with calculating only the MS half of $m \cdot m^{-1}$, plus a couple of guard digits to get an approximate quotient, to be rounded to the nearest integer.

Unfortunately there is no trivial modification of the algorithm to handle even moduli directly, because at halving only an integer multiple of the modulus can be added without changing the result, and only adding an odd number can turn odd intermediate values to even. Fortunately, the only time we need to handle even moduli in cryptography is at RSA key generation, which is so slow anyway (requiring thousands of modular multiplications for the primality tests), that this black box workaround does not cause a noticeable difference in processing time.

An alternative was to perform the full extended GCD algorithm, calculating both factors c and d: $[g, c, d] = x\text{CGD}(m, a)$, such that the greatest common divisor $g = c \cdot m + d \cdot a$ [5]. It would need extra storage for two factors, which are constantly updated during the course of the algorithm and it is also slower than applying the method above transforming the result of the modular inverse algorithm with swapped parameters.

3.1.3. Justification

The algorithm starts with U $= m$, V $= a$, R $= 0$, S $= 1$. In the course of the algorithm U and V are decreased, keeping GCD(U, V) $=$ GCD(m, a) true. The algorithm reduces U and V until V $= 0$ and U $=$ GCD(m, a): if one of U or V is even, it can be replaced by its half, since GCD(m, a) is odd. If both are odd, the larger one can be replaced by the even U $-$ V, which then can be decreased by halving, leading eventually to 0. The binary length of the larger of U and V is reduced by at least one bit, guaranteeing that the procedure terminates in at most $\|a\| + \|m\|$ iterations.

At termination of the algorithm V $= 0$ otherwise a length reduction was still possible. U $=$ GCD(U, 0) $=$ GCD(m, a). Furthermore, the calculations maintain the following two congruencies:

$$\text{U} \equiv Ra \bmod m, \qquad \text{V} \equiv Sa \bmod m. \qquad (5)$$

Having an odd modulus m, at the step halving U we have two cases. When R is even: U/2 $\equiv$ (R/2) $\cdot\, a \bmod m$, and when R is odd: U/2 $\equiv$ ((R $+ m$)/2) $\cdot\, a \bmod m$. The algorithm assigns these to U and R. Similarly for V and S, and with their new values, (5) remains true.

The difference of the two congruencies in (5) gives U $-$ V $\equiv$ (R $-$ S) $\cdot\, a \bmod m$, which ensures that at the subtraction steps (5) remains true after updating the corresponding variables: U or V $\leftarrow$ U $-$ V, R or S $\leftarrow$ R $-$ S. Choosing $+m$ or $-m$, as discussed above, guarantees that R and S does not grow larger than $2m$, so at the end we can just add or subtract m to make $0 <$ R $< m$. If U $= 1 =$ GCD(m, a), we get $1 \equiv Ra \bmod m$, and R is of the right magnitude, so R $= a^{-1} \bmod m$.

3.1.4. Plus-minus: algorithm RS+−

There is a very simple modification often used for the right-shift algorithm [14]: for the odd U and V check, if U + V has 2 trailing 0 bits, otherwise we know that U − V does. In the former case, if U + V is of the same length as the larger of them, the shift operation reduces the length by 2 bits from this larger length, otherwise by only one bit (as before with the rigid subtraction steps). It means that the length reduction is sometimes improved, so the number of iterations decreases.

Unfortunately, this reduction is not large, only 15% (half of the time the reduction was by at least 2 bits, anyway, and longer shifts are not affected either), but it comes almost for free. Furthermore, R and S need more halving steps, and these get a little more expensive (at least one of the halving steps needs an addition of m), so the RS+− algorithm is not faster than RS1.

3.1.5. Double plus-minus: algorithm RS2+−

The plus-minus reduction can be applied also to R and S (Algorithm 2). In the course of the algorithm they get halved, too. If one of them happens to be odd, m is added or subtracted to make them even before the halving. The plus-minus trick on them ensures that the result has at least 2 trailing 0 bits. It provides a speedup, because most of the time we had exactly two divisions by 2 (shift right by two), and no more than one addition/subtraction of m is now necessary.

3.1.6. Delayed halving: algorithm RSDH

The variables R and S get almost immediately of the same length as m, because, when they are odd, m is added to them to allow halving without remainder. We can delay these add-halving steps, by doubling the other variable instead. When R should be halved we double S, and vice versa. Of course, a power-of-2 spurious factor is introduced to the computed GCD, but keeping track of the exponent a final correction step will fix R by the appropriate number of halving or add-halving steps. (This technique is similar to the Montgomery inverse computation published in [15] and sped up for computers in [16], but the correction steps differ.) It provides an acceleration of the algorithm by 24–38% over RS1, due to the following.

```
U ← $m$; V ← $a$;
R ← 0; S ← 1;
$Q = m \bmod 4$;
while ($V_0 = 0$) {          V ← V/2;
      if ($S_0 = 0$)          S ← S/2;
      else if (S > $m$)       S ← (S − $m$)/2;
      else                    S ← (S + $m$)/2;
}
Loop {    // U, V odd
    if (U > V) {
      if ($U_1 = V_1$)
          U ← U + V;    R ← R + S;
      else
          U ← U − V;    R ← R − S;
      U ← U/4;          T ← R mod 4;
      if (T = 0)        R ← R/4;
      if (T = 2)        R ← (R + 2$m$)/4;
      if (T = $Q$)      R ← (R − $m$)/4;
      else              R ← (R + $m$)/4;
      while ($U_0 = 0$)  {U ← U/2;
          if ($R_0 = 0$)    R ← R/2;
          else if (R > $m$) R ← (R − $m$)/2;
          else              R ← (R + $m$)/2; }
    else {
      if ($U_1 = V_1$)
          V ← V + U;     S ← S + R;
      else
          V ← V − U;     S ← S − R;
          if (V = 0)     break;
      V ← V/4;           T ← S mod 4;
      if (T = 0)         S ← S/4;
      if (T = 2)         S ← (S + 2$m$)/4;
      if (T = $Q$)       S ← (S − $m$)/4;
      else               S ← (S + $m$)/4;
      while ($V_0 = 0$)   {V ← V/2;
          if ($S_0 = 0$)     S ← S/2;
          else if (S > $m$)  S ← (S − $m$)/2;
          else               S ← (S + $m$)/2; }
}
if (U > 1) return 0;           // no inverse
if (R ≥ $m$) R ← R − $m$;
if (R < 0) R ← R + $m$;
return R;                      //$a^{-1} \bmod m$
```

Algorithm 2: Double plus-minus right-shift binary algorithm.

(1) R and S now increase gradually, so their average length is only half as it was in RS1.
(2) The final halving steps are performed only with R. The variable S needs not be fixed, being only an internal temporary variable.
(3) At the final halving steps more short shifts can be combined to longer shifts, because they are not confined by the amount of shifts performed on U and V in the course of the algorithm.

Note 1. R and S are almost always of different lengths, and so their difference is not longer than the longer of R and S. Consequently, their lengths do not increase faster than what the shifts cause.

Note 2. It does not pay to check, if R or S is even, in the hope that some halving steps could be performed until the involved R or S becomes odd, and so speeding up the final correction, because they are already odd in the beginning (easily proved by induction).

3.1.7. *Combined speedups: algorithm RSDH+−*

The second variant of the plus-minus trick and the delayed halving trick can be combined, giving the fastest of the presented right-shift modular algorithms. It is 43–60% faster than algorithm RS1 (which is 30% faster than the traditional implementation RS), but still slower on most computational platforms than the left-shift and shifting Euclidean algorithms, discussed below.

3.2. *Binary left-shift modular inverse: algorithm LS1*

The left-shift binary modular inverse algorithm (similar to the variant of Lórencz [17]) is described in Algorithm 3. It keeps the temporary variables U and V aligned to the left, such that a subtraction clears the leading bit(s). Shifting the result left until the most significant bit is again in the proper position restores the alignment. The number of known trailing 0 bits increases, until a single 1 bit remains, or the result is 0 (indicating that there is no inverse). As before, keeping 2 internal variables R and S updated, the modular inverse is calculated.

Here u and v are single-word variables, counting how many times U and V were shifted left, respectively. They tell at least how many trailing zeros the corresponding U and V long integers have, because we always add/subtract to the one, which has fewer known zeros and then shift left, increasing the number of trailing zeros. 16 bit words for u and v allow us working with any operand length less than 64 Kb, enough for all cryptographic applications in the foreseeable future. Knowing the values of u and v also helps speeding up the calculations, because we need not process the known least significant zeros.

3.2.1. *Justification*

The reduction of the temporary variables is now done by shifting left the intermediate results U and V, until they have their MS bits in the designated nth bit position (which is the MS position of the larger of the original operands). Performing a subtraction clears this bit, reducing the binary length. The left shifts introduce spurious factors, 2^k, for the GCD, but tracking the number of trailing 0 bits (u and v) allows the determination of the true GCD. (For a rigorous proof see [17].)

We start with $U = m$, $V = a$, $R = 0$, $S = 1$, $u = v = 0$. In the course of the algorithm there will be at least u and v trailing 0 bits in U and V, respectively. In the beginning

$$\mathrm{GCD}\left(U/2^{\min(u,v)}, V/2^{\min(u,v)}\right) = \mathrm{GCD}(m, a). \qquad (6)$$

If U or V is replaced by U − V, this relation remains true. If both U and V had their most significant (nth) bit = 1, the

```
U ← $m$; V ← $a$;
R ← 0; S ← 1;
$u$ ← 0; $v$ ← 0;
while (($|U| \neq 2^u$)   &&   ($|V| \neq 2^v$)) {
    if ($|U| < 2^{n-1}$) {
        U ← 2U; $u$ ← $u+1$;
        if ($u > v$) R ← 2R;
        else         S ← S/2;
    }
    else if ($|V| < 2^{n-1}$) {
        V ← 2V; $v$ ← $v+1$;
        if ($v > u$)   S ← 2S;
        else           R ← R/2;
    }
    else // $|U|, |V| \geq 2^{n-1}$
      if (sign(U) = sign(V))
        if ($u \leq v$)
          {U ← U − V; R ← R − S; }
        else
          {V ← V − U; S ← S − R; }
      else // sign(U) ≠ sign(V)
        if ($u \leq v$)
          {U ← U + V; R ← R + S; }
        else
          {V ← V + U; S ← S + R; }
    if (U = 0 || V = 0) return 0; }
if ($|V| = 2^v$) {R ← S; U ← V; }
if (U < 0)
    if (R < 0) R ← −R;
    else R ← $m$ − R;
if (R < 0) R ← $m$ + R;
return R; // $a^{-1} \bmod m$
```

Algorithm 3: Left-shift binary algorithm.

above subtraction clears it. We chose the one from U and V to be updated, which had the smaller number of trailing 0 bits, say it was U. U then gets doubled until its most significant bit gets to the nth bit position again, and u, the number of trailing 0's, is incremented in each step.

If $u \geq v$ was before the doubling, $\min(u, v)$ does not change, but U doubles. Since GCD(m, a) is odd (there is no inverse if it is not 1), GCD$(2 \cdot \text{U}/2^{\min(u,v)}, \text{V}/2^{\min(u,v)})$ = GCD(m, a) remains true. If $u < v$ was before the doubling, $\min(u, v)$ increases, leaving $\text{U}/2^{\min(u,v)}$ unchanged. The other parameter $\text{V}/2^{\min(u,v)}$ was even, and becomes halved. It does not change the GCD, either.

In each subtraction-doubling iteration either u or v (the number of trailing known 0's) is increased. U and V are never longer than n-bits, so u and $v \leq n$, and eventually a single 1 bit remains in U or V (or one of them becomes 0, showing that GCD$(m, a) > 1$). It guarantees that the procedure stops in at most $\|a\| + \|m\|$ iterations, with U or V = 2^{n-1} or 0.

In the course of the algorithm,

$$\text{U}/2^{\min(u,v)} \equiv Ra \bmod m, \qquad \text{V}/2^{\min(u,v)} \equiv Sa \bmod m. \quad (7)$$

At subtraction steps $(\text{U}-\text{V})/2^{\min(u,v)} \equiv (\text{R}-\text{S}) \cdot a \bmod m$, so (7) remains true after updating the corresponding variables: U or V ← U − V, R or S ← R − S.

At doubling U and incrementing u, if $u < v$ was before the doubling, $\min(u, v)$ increases, so $\text{U}/2^{\min(u,v)}$ and R remains unchanged. $\text{V}/2^{\min(u,v)}$ got halved, so it is congruent to $(\text{S}/2) \cdot a \bmod m$, therefore S has to be halved to keep (7) true. This halving is possible (V is even), because S has at least $v - u$ trailing 0's (can be proved by induction).

At doubling U and incrementing u, if $u \geq v$ was before the doubling, $\min(u, v)$ does not change. To keep (7) true R has to be doubled, too (which also proves that it has at least $v - u$ trailing 0's).

Similar reasoning shows the correctness of handling R and S when V is doubled.

At the end we get either U $= 2^u$ or V $= 2^v$, so one of $\text{U}/2^{\min(u,v)}$ or $\text{V}/2^{\min(u,v)}$ is 1, and GCD(m, a) is the other one. If the inverse exists, GCD$(m, a) = 1$ and we get from (7) that either $1 \equiv Ra \bmod m$ or $1 \equiv Sa \bmod m$. After making R or S of the right magnitude, it is the modular inverse $a^{-1} \bmod m$.

Another induction argument shows that R and S do not become larger than $2m$ in the course of the algorithm, otherwise the final reduction phase of the result to the interval $[1, m-1]$ could take a lot of calculations.

3.2.2. *Best left shift: algorithm LS3*

The plus-minus trick does not work with the left-shift algorithm: addition never clears the MS bit. If U and V are close, a subtraction might clear more than one MS bits, otherwise one could try 2U − V and 2V − U for the cases when 2U and V or 2V and U are close. (With the nth bit = 1 other two's power linear combinations, which can be calculated with only shifts, do not help.) Looking at only a few MS bits, one can determine which one of the 3 tested reductions is expected to give the largest length decrease (testing 3 reduction candidates is the reason to call the algorithm LS3). We could often clear extra MS bits this way. In general microprocessors the gain is not much, because computing $2x - y$ could take 2 instructions instead of one for $x - y$, but memory load and store steps can still be saved. With hardware for shifted operand fetch the doubling comes for free, giving a larger speedup.

3.3. *Shifting Euclidean modular inverse: algorithms SE*

The original Euclidean GCD algorithm replaces the larger of the two parameters by subtracting the largest number of times the smaller parameter keeping the result nonnegative: $x \leftarrow x - [x/y] \cdot y$. For this we need to calculate the quotient $[x/y]$ and multiply it with y. In this paper we do not deal with algorithms, which perform division or multiplication. However, the Euclidean algorithm works with smaller coefficients $q \leq [x/y]$, too: $x \leftarrow x - q \cdot y$. In particular, we can choose q to be the largest power of 2, such that $q = 2^k \leq [x/y]$. The reductions can be performed with only shifts and subtractions, and they still clear the most significant bit of x, so the resulting algorithm will terminate in a reasonable number of iterations. It is well known (see [12]) that for random input, in the course of the algorithm, most of the time $[x/y] = 1$ or 2, so the shifting Euclidean algorithm performs only slightly

```
if ($a < m$)
  {$U \leftarrow m$; $V \leftarrow a$;
   $R \leftarrow 0$; $S \leftarrow 1$; }
else
  {$V \leftarrow m$; $U \leftarrow a$;
   $S \leftarrow 0$; $R \leftarrow 1$; }
while ($\|V\| > 1$) {
    $f \leftarrow \|U\| - \|V\|$
    if (sign(U) = sign(V))
        {$U \leftarrow U - (V \ll f)$;
         $R \leftarrow R - (S \ll f)$; }
    else
        {$U \leftarrow U + (V \ll f)$;
         $R \leftarrow R + (S \ll f)$; }
    if ($\|U\| < \|V\|$)
        {$U \leftrightarrow V$; $R \leftrightarrow S$; }
}
if ($V = 0$) return 0;
if ($V < 0$) $S \leftarrow -S$;
if ($S > m$) return $S - m$;
if ($S < 0$) return $S + m$;
return S; // $a^{-1} \bmod m$
```

Algorithm 4: Shifting Euclidean algorithm.

more iterations than the original, but avoids multiplications and divisions. See Algorithm 4.

Repeat the above reduction steps until $V = 0$ or ± 1, when $U = \mathrm{GCD}(m, a)$. If $V = 0$, there is no inverse, so we return 0, which is not an inverse of anything. (The pathological cases like $m = a = 1$ need special handling, but these do not occur in cryptography.)

In the course of the algorithm two auxiliary variables, R and S are kept updated. At termination S is the modular inverse, or the negative of it, within $\pm m$.

3.3.1. Justification

The algorithm starts with $U = m$, $V = a$, $R = 0$, $S = 1$. If $a > m$, swap (U, V) and (R, S). U always denotes the longer of the just updated U and V. During the course of the algorithm U is decreased, keeping $\mathrm{GCD}(U, V) = \mathrm{GCD}(m, a)$ true. The algorithm reduces U, swaps with V when $U < V$, until $V = \pm 1$ or 0 : U is replaced by $U - 2^k V$, with such a k, that reduces the length of U, leading eventually to 0 or ± 1, when the iteration can stop. The binary length $\|U\|$ is reduced by at least one bit in each iteration, guaranteeing that the procedure terminates in at most $\|a\| + \|m\|$ iterations.

At termination of the algorithm either $V = 0$ (indicating that $U = 2^k V$ was beforehand, and so there is no inverse) or $V = \pm 1$, otherwise a length reduction was still possible. In the later case $1 = \mathrm{GCD}(|U|, |V|) = \mathrm{GCD}(m, a)$. Furthermore, the calculations maintain the following two congruencies:

$$U \equiv Ra \bmod m, \qquad V \equiv Sa \bmod m. \tag{8}$$

The weighted difference of the two congruencies in (8) gives $U - 2^k V \equiv (R - 2^k S) \cdot a \bmod m$, which ensures that at the reduction steps (8) remains true after updating the corresponding variables: $U \leftarrow U - 2^k V$, $R \leftarrow R - 2^k S$. As in the proof of correctness of the original extended Euclidean algorithm, we can see that $|R|$ and $|S|$ remain less than $2m$, so at the end we fix the sign of S to correspond to V, and add or subtract m to make $0 < S < m$. Now $1 \equiv Sa \bmod m$, and S is of the right magnitude, so $S = a^{-1} \bmod m$.

3.3.2. Best-shift Euclidean modular inverse: algorithm SE3

We can employ a similar speedup technique for the shifting Euclidean algorithm as with the left-shift algorithm LS3. If U and $2^k V$ are close, the shift subtraction might clear more than one MS bits, otherwise one could try $U - 2^{k-1} V$ and $U - 2^{k+1} V$. (With k being the length difference. Other two's power linear combinations cannot clear more MS bits.) Looking at only a few MS bits one can determine which one of the 3 tested reductions is expected to give the largest (length) decrease. (Testing 3 reduction candidates is the reason to call the algorithm SE3). We could often clear extra MS bits this way. This technique gives about 14% reduction in the number of iterations, and a similar speedup on most computational platforms, because the shift operation takes the same time, regardless of the amount of shift (except when it is 0).

We have a choice: how to rank the expected reductions. In the SE3 code we picked the largest expected length reduction, because it is the simplest in hardware. Another possibility was to choose the shift amount, which leaves the smallest absolute value result. It is a little more complex, but gives about 0.2% speed increase.

4. SIMULATION TEST RESULTS

The simulation code was written in C, developed in MS Visual Studio 6. It is available at *http://www.hars.us/SW/ModInv.c*. GMP Version 4.1.2, the GNU multiprecision arithmetic library [3] was used for the long integer operations and for verifying the results. It is linked as an MS Windows DLL, available also at *http://www.hars.us/SW/gmp-dll.zip*.

We executed 1 million calls of each of the many variants of the modular inverse algorithms with 14 different lengths in the range of 16–1024 bit random inputs, so the experimental complexity results are expected to be accurate within 0.1–0.3% (central limit theorem) at every operand length. The performed operations and their costs were counted separately for different kind of operations. Table 1 contains the binary costs of the additions and shifts the corresponding modular inverse algorithms performed, and the number of iterations and the number of shifts with the most frequent lengths. (Multiple shifts are combined together.) The computed curves fit to the data with less than 1% error at any operand length.

The *right-shift algorithms* are the slowest, because they halve two auxiliary variables (R, S) and if they happen to be

Table 1

Algorithm	Right shift					Left shift		Shift Euclidean	
Steps/bit	RS1	RS+−	RS2+−	RSDH	RSDH+−	LS1	LS3	SE	SE3
Iterations	$0.7045n$	$0.6115n$	$0.6115n$	$0.7045n$	$0.6115n$	$0.7650n$	$0.6646n$	$0.7684n$	$0.6744n$
UV shift cost	$0.3531n^2$ $-1.2200n$	$0.3065n^2$ $-1.1891n$	$0.3065n^2$ $-1.1891n$	$0.3531n^2$ $-1.2200n$	$0.3065n^2$ $-1.1891n$	$0.3834n^2$ $-0.8836n$	$0.3967n^2$ $-0.8435n$	$0.3101n^2$ $-1.0646n$	$0.2708n^2$ $-0.8742n$
RS shift cost	$1.0592n^2$ $-4.9984n$	$1.2259n^2$ $-5.2592n$	$0.9808n^2$ $-5.1720n$	$0.9241n^2$ $-3.3945n$	$0.8021n^2$ $-3.3794n$	$0.5300n^2$ $-4.9665n$	$0.5558n^2$ $-5.1855n$	$0.3101n^2$ $-2.9784n$	$0.2708n^2$ $-2.5787n$
Total shift cost	$1.4123n^2$ $-6.2184n$	$1.5324n^2$ $-6.4483n$	$1.2873n^2$ $-6.3611n$	$1.2772n^2$ $-4.6145n$	$1.1086n^2$ $-4.5685n$	$0.9134n^2$ $-5.8501n$	$0.9525n^2$ $-6.0290n$	$0.6202n^2$ $-4.0430n$	$0.5416n^2$ $-3.4529n$
UV subtract cost	$0.3531n^2$ $+0.2658n$	$0.3065n^2$ $+0.2967n$	$0.3065n^2$ $+0.2967n$	$0.3531n^2$ $+0.2658n$	$0.3065n^2$ $+0.2967n$	$0.3835n^2$ $+0.4377n$	$0.3331n^2$ $+0.5942n$	$0.3851n^2$ $+0.4276n$	$0.3380n^2$ $+0.4958n$
RS subtract cost	$1.4123n^2$ $-4.8065n$	$1.5325n^2$ $-4.8844n$	$1.2873n^2$ $-4.5004n$	$0.9241n^2$ $-1.4559n$	$0.8021n^2$ $-0.7786n$	$0.3834n^2$ $-1.0101n$	$0.3331n^2$ $-0.9160n$	$0.3851n^2$ $-1.0331n$	$0.3380n^2$ $-0.7125n$
Total subtract cost	$1.7654n^2$ $-4.5407n$	$1.8390n^2$ $-4.5877n$	$1.5938n^2$ $-4.2037n$	$1.2772n^2$ $-1.1901n$	$1.1086n^2$ $-0.4819n$	$0.7669n^2$ $-0.5724n$	$0.6662n^2$ $-0.3218n$	$0.7702n^2$ $-0.6055n$	$0.6760n^2$ $-0.2167n$
Complexity at 0 cost shift	$1.7654n^2$	$1.8390n^2$	$1.5938n^2$	$1.2772n^2$	$1.1086n^2$	$0.7669n^2$	$0.6662n^2$	$0.7702n^2$	$0.6750n^2$
Complexity at 1/4 add cost shift	$2.1185n^2$	$2.2221n^2$	$1.9156n^2$	$1.5965n^2$	$1.3858n^2$	$0.9953n^2$	$0.9043n^2$	$0.9253n^2$	$0.8114n^2$
Complexity at 1 add cost shift	$3.1777n^2$	$3.3714n^2$	$2.8811n^2$	$2.5544n^2$	$2.2172n^2$	$1.6803n^2$	$1.6187n^2$	$1.3904n^2$	$1.2176n^2$
UV shifts by 1	$0.3522n$	—	—	$0.3522n$	—	$0.1983n$	$0.1977n$	$0.2576n$	$0.2143n$
UV shifts by 2	$0.1761n$	$0.3058n$	$0.3058n$	$0.1761n$	$0.3058n$	$0.2463n$	$0.2388n$	$0.1705n$	$0.1573n$
UV shifts by 3	$0.0881n$	$0.1529n$	$0.1529n$	$0.0881n$	$0.1529n$	$0.1516n$	$0.1778n$	$0.0927n$	$0.0831n$
Longer UV shifts	$0.0881n$	$0.1529n$	$0.1529n$	$0.0881n$	$0.1529n$	$0.1689n$	$0.1772n$	$0.0980n$	$0.0857n$
RS shifts by 1	$0.7925n$	$0.7644n$	$0.3364n$	$0.6375n$	—	$0.5202n$	$0.5395n$	$0.2576n$	$0.2143n$
RS shifts by 2	$0.1982n$	$0.3440n$	$0.4816n$	$0.3188n$	$0.5534n$	$0.3142n$	$0.3313n$	$0.1705n$	$0.1573n$
RS shifts by 3	$0.0495n$	$0.0860n$	$0.1204n$	$0.1594n$	$0.2767n$	$0.1280n$	$0.1413n$	$0.0927n$	$0.0831n$
Longer RS shifts	$0.0165n$	$0.0287n$	$0.0401n$	$0.1594n$	$0.2767n$	$0.0952n$	$0.0968n$	$0.0980n$	$0.0857n$

odd, m is added or subtracted first, to make them even for the halving. Theoretical arguments and also our computational experiments showed that they are too slow at *digit-serial* arithmetic. They were included in the discussions mainly, because there are surprisingly many systems deployed using some variant of the right-shift algorithm, although others are much better.

The addition steps are not needed in the *left-shift* or in the *shifting Euclidean* algorithms. In all three groups of algorithms the length of U and V decreases bit-by-bit in each iteration, and in the left-shift and shifting Euclidean algorithms the length of R and S increases steadily from 1. In the right-shift case they get very soon as long as m, except in the delayed halving variant. In the average, the changing lengths roughly halve the work on those variables. Also, the necessary additions of m in the original right-shift algorithms prevent aggregation of the shift operations of R and S. On the other hand, in the other algorithms (including the delayed halving right-shift algorithm) we can first determine by how many bits we have to shift all together in that phase. In the left-shift algorithms, dependent on the relative magnitude of u and v, we need only one or two shifts by multiple bits, in the shifting Euclidean algorithm only one. This shift aggregation saves work at longer shifts than the most common lengths of 1 or 2.

On the other hand, the optimum shift lengths in the left-shift and shifting Euclidean algorithms are only estimated from the MS bits. They are sometimes wrong, while in the right-shift algorithm only the LS bits play a role, so the optimum shift lengths can always be found exactly. Accordingly, the right-shift algorithms perform slightly fewer iterations (8.6–10%), but the large savings in additions in the other algorithms offset these savings.

4.1. Software running time comparisons

We did not measure execution times of SW implementations, because of the following reasons.

(1) The results are very much dependent on the characteristics of the *hardware* platforms (word length, instruction timings, available parallel instructions, length and function of the instruction pipeline, processor versus memory speed, cache size and speed, number of levels of cache memory, page fault behavior, etc).
(2) The results also depend on the *operating system* (multitasking, background applications, virtual/paging memory handling, etc).
(3) The results are dependent on the *code*, the *programming language*, and the *compiler*. For example, GMP [3] uses hand optimized assembler macros, and any other SW written in a higher level language is necessarily disadvantaged, like at handling carries.

In earlier publications running time measurements were reported, like in [18] Jebelean gave software execution time measurements of GCD algorithms on a DEC computer of RISC architecture. Our measurements on a 3 GHz Intel Pentium PC running Windows XP gave drastically different speed ratios. This large uncertainty was the reason why we decided to count the number of specific operations and sum up their time consumption dependent on the operand lengths, instead of the much easier running time measurements. This way the actual SW running time can be well estimated on many different computational platforms.

4.2. Notes on the simulation results

(i) The number of the different UV shifts, together, is the number of iterations, since there is one combined shift in each iteration.
(ii) In the left-shift algorithms the sum of RS shifts is larger than the number of iterations, because some shifts may cause the relationship between u and v to change, and in this case there are 2 shifts in one iteration.
(iii) In [19] there are evidences cited that the binary right-shift GCD algorithm performs $A \cdot \log 2^{\|m\|}$ iterations, with $A = 1.0185\ldots$. The RS1 algorithm performs the same number of iterations as the binary right-shift GCD algorithm. Our experiments gave a very similar (only 0.2% smaller) result: $A' = 0.7045/\log 2 = 1.0164\ldots$.

In Table 1 we listed the coefficients of the dominant terms of the best fit polynomials to the time consumption of the algorithms, in 3 typical computational models.

(1) Shifts execute in a constant number of clock cycles

Algorithm LS3 is the fastest ($0.6662n^2$), followed by SE3 ($0.6750n^2$), with only a 1.3% lag. The best right-shift algorithm is RSDH+−, which is 1.66 times slower ($1.1086n^2$).

(2) Shifts are 4 times faster than add/subtracts

Algorithm SE3 is the fastest ($0.8114n^2$), followed by LS3 ($0.9043n^2$), within 14%. The best right-shift algorithm (RSDH+−) is 1.71 times slower ($1.3858n^2$).

(3) Shifts and add/subtracts take the same time

Again, algorithm SE3 is the fastest ($1.2176n^2$), followed by SE ($1.3904n^2$), within 14%. The best right-shift algorithm (RSDH+−) is 2.37 times slower ($2.8804n^2$).

Interestingly the plus-minus algorithm RS+−, which only assures that U or V are reduced by at least 2 bits, performs fewer iterations, but the overall running time is not improved. When R and S are also handled this way, the running time improves. It shows that speeding up the (R, S) halving steps is more important than speeding up the (U, V) reduction steps, because the later reduction steps operate on diminishing length numbers, while the (R, S) halving works mostly on more costly, full length numbers.

4.3. Performance relative to digit-serial modular multiplication

Of course, the speed ratio of the modular inverse algorithms relative to the speed of the modular multiplications depends on the computational platform and the employed multiplication algorithm. We consider quadratic time modular multiplications, like Barrett, Montgomery, or school multiplication with division-based modular reduction (see [5]). With operand lengths in cryptography subquadratic time modular multiplications (like Karatsuba) are only slightly faster, more often they are even slower than the simpler quadratic time algorithms (see [3]).

If there is a hardware multiplier, which computes products of d-bit digits in c clock cycles, a modular multiplication takes $T = 2c \cdot (n/d)^2 + O(n)$ time alone for computing the digit products [11]. In DSP-like architectures (load, shift, and add instructions performed parallel to multiplications) the time complexity is $2c \cdot (n/d)^2$. Typical values are

(i) $d = 16, c = 4$: $T = n^2/32 \approx 0.031n^2$,
(ii) $d = 32, c = 12$: $T = 3n^2/128 \approx 0.023n^2$.

The fastest of the presented modular inverse algorithm on *parallel shift-add* architecture takes $0.666n^2$ bit operations, which needs to be divided by the digit size (processing d bits together in one addition). For the above two cases we get $0.042n^2$ and $0.021n^2$ running times, respectively. These values are very close to the running time of *one* modular multiplication.

The situation is less favorable if there are *no parallel* instructions. The time a multiplication takes is dominated by computing the digit products. Additions and register manipulations are faster. On these platforms computing the modular inverse takes almost twice as much time than with parallel add and shift instructions. Consequently, computing the modular inverse without parallel instructions takes about *twice* as much time as a modular multiplication. Still, in case of elliptic curve cryptography the most straightforward (affine) point representation and direct implementation of the point addition is the best (computation with the projective, Jacobian and Chudnovsky-Jacobian coordinates are slower, see [9]).

It is interesting to note that modular division ($p \cdot q^{-1}$) can be performed faster than 3 modular multiplications. Similar results were presented in [7, 20] for polynomials of practical lengths, showing that even in extension fields GF(p^k), elliptic curve points are best represented in affine coordinates.

4.4. Performance relative to parallel adder-based modular multiplication

When very long adders are implemented in hardware, repeated shift-add steps can perform multiplications in linear time. To prevent the partial results from growing too large, interleaved modular reduction is performed. Scanning the bits of the second multiplicand from left to right, when a 1 bit is found, the first multiplicand is added in the appropriate position to the partial result r. If it gets too long: $\|r\| > \|m\|$, it is reduced by subtracting the modulus m. These add-subtract steps are usually done in the same clock cycle, resulting in performing an n-bit modular multiplication in n clock cycles.

In these kinds of hardware architectures the speed of the different modular inverse algorithms becomes very close, because there is no advantage of having additions on diminishing length operands. An average iteration reduces the length of the longer operand by about 1.4 bits, so the left- and right-shift algorithms do not differ much, in how many shift steps can be combined into one longer shift. The plus-minus right-shift algorithm has the smallest number of iterations, its delayed halving variant can combine the largest number of shifts, so its running time becomes very close to that of the shifting Euclidean algorithm.

In each iteration the RSDH+− modular inverse algorithm needs to shift one of U or V, and double R or S the same many times, which give about the same amount of work as the modular multiplication performs, maybe even less. At the end we need to add-halve R, which makes the modular inverse slightly slower than *one* modular multiplication, but still faster than two.

4.5. Testing relative primality

We can simplify all of our shifting modular inverse algorithms if we only want to know whether the two arguments x, y are relative primes: leaving out all the calculations with R and S. In this case all the plus-minus right-shift algorithms become the same, so the simplest one, RS+− is the best, with $0.3065n^2$ cost of bit shifts and the same for subtractions. All together it is $0.6130n^2$. SE3 is still slightly better, with a running time of $0.6088n^2$. The modified left-shift algorithm LS3 takes $0.3967n^2$ clock cycles for the shift operations, and $0.3331n^2$ clock cycles for the subtract operations, which is only 9–19% more. When, in an application, not only relative primality has to be tested, but modular inverses have to be calculated as well, this little speed advantage might not justify the implementation of 2 different algorithms, so LS3 or SE3 should be used for both purposes (without computing R and S if not needed).

5. FURTHER OPTIMIZATIONS POSSIBILITIES

There are countless possibilities to speed up the presented algorithms a little further. For example, when U and V become small (short), a table lookup could immediately finish the calculations. If only one of them becomes small, or there is a large difference of the lengths of U and V, we could perform a different algorithmic step, which is best tuned to this case on the particular computing platform. (Most of the time it is a traditional Euclidean reduction step.) We tried hundreds of ideas like these, but the little acceleration did not justify the increased complexity.

Some of the presented speedup methods could have been applied already somewhere in the huge literature of algorithmic optimization, but we could not find the wining combinations of these optimization techniques for the modular inverse problem published anywhere. Many modifications accelerate one part of the algorithm while they slow down—or even invalidate—other parts. We investigated hundreds of algorithmic changes, but only discussed here the original algorithms and those optimizations, which led to the largest speedups.

5.1. Working on the ends

On some computational platforms speed increase can be achieved with delayed full update of the variables. See, for example [18, 21] or [22]. It means working with MS and/or LS digits only, as long as we can determine the necessary reduction steps, and fix the rest of the digits only when more precision is needed. Speedup is achieved by the reduced number of data fetch, and combined update operations on the middle digits. Unfortunately, the resulting algorithms are much more complex, less suitable for direct hardware implementations and the combined operations involve multiplications, what we wanted to avoid. In our computational model data load-store operations are free, so having fewer of them does not provide any speed advantage.

5.2. Hybrid algorithms

The right-shift and the shifting Euclidean algorithms operate on the opposite ends of the numbers. At least when the modulus is odd, the reduction steps could be intermixed: check on a few bits on the appropriate end of the intermediate values which algorithm is expected to reduce the lengths the most, and perform one step of it. However, the right-shift algorithms are so much slower, that the corresponding reduction is almost never expected to be faster than the shifting Euclidean reduction. This way we could not achieve any significant speedup, but the algorithms became complicated and convoluted.

5.3. *Modular division*

If the initialization of S $\leftarrow$ 1 is replaced by S $\leftarrow d$, we get da^{-1} as the result, as described in [7] for polynomials. In case the modular inverse is only needed once, and it is multiplied by another number, we could save that multiplication, like in elliptic curve cryptography. If the inverse is reused many times, like at signed digit exponentiation [8], this trick does not improve performance.

We start with a full length $\mathrm{S}(= d)$ instead of length 1, so (S, R) do not gradually increase from length 1, but start at length n. Further steps are necessary to prevent them to grow too large. These more than double the total work updating (S, R) (but not (U, V)) at the left-shift and shifting Euclidean algorithms, all together 50–100% increase. The right-shift algorithms do not change much, so modular divisions significantly reduce their performance lag. In general, it only pays doing divisions this way, when the underlying modular inverse algorithm is much faster than two modular multiplications (making a modular division faster than 3 modular multiplications).

NOTATIONS

(i) Modular inverse: a^{-1}, the smallest positive integer for integer a, such that $a \cdot a^{-1} = 1 \bmod m$.
(ii) GCD: greatest common divisor of integers.
(iii) xGCD or *extended* GCD: the algorithm calculating g and also two factors c and d: $[g, c, d] = x\mathrm{CGD}(x, y)$, such that the greatest common divisor of x and y is g, and $g = c \cdot x + d \cdot y$.
(iv) MS/LS bits: the most/least significant bits of binary numbers.
(v) $\|m\|$ the number of bits in the binary representation of integer m, its binary length.
(vi) $m = \{m_{n-1}, \ldots, m_1, m_0\} = m_{n-1,\ldots,0} = \Sigma_{i=0,\ldots,n-1} 2^i m_i$, where the bits $m_i \in \{0, 1\}$ of the integer m form its *binary representation*.

ACKNOWLEDGMENT

The author thanks the anonymous reviewers for suggesting many corrections, improvements, and the reorganization of the original manuscript.

REFERENCES

[1] M. Joye and P. Paillier, "GCD-free algorithms for computing modular inverses," in *Proceedings of 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '03)*, vol. 2779 of *Lecture Notes in Computer Science*, pp. 243–253, Cologne, Germany, September 2003.

[2] A. Schönhage and V. Strassen, "Schnelle Multiplikation großer Zahlen," *Computing*, vol. 7, pp. 281–292, 1971.

[3] GNU Multiple Precision Arithmetic Library manual, http://www.swox.com/gmp/gmp-man-4.1.2.pdf.

[4] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer, New York, NY, USA, 2004.

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

[6] L. Hars, "Fast truncated multiplication and its applications in cryptography," in *Proceedings of 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '05)*, Edinburgh, Scotland, August 2005.

[7] S. C. Shantz, "From Euclid's GCD to Montgomery multiplication to the great divide," Tech. Rep. TR-2001-95, Sun Microsystems Laboratories, Santa Clara, Calif, USA, June 2001.

[8] J. Jedwab and C. J. Mitchell, "Minimum weight modified signed-digit representations and fast exponentiation," *Electronics Letters*, vol. 25, no. 17, pp. 1171–1172, 1989.

[9] H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates," in *Proceedings of International Conference on the Theory and Applications of Cryptology and Information Security, Advances in Cryptology (ASIACRYPT '98)*, K. Ohta and D. Pei, Eds., vol. 1514 of *Lecture Notes in Computer Science*, pp. 51–65, Beijing, China, October 1998.

[10] M. D. Ercegovac and T. Lang, *Digital Arithmetic*, chapter 2, Morgan Kaufmann, San Francisco, Calif, USA, 2004.

[11] L. Hars, "Long modular multiplication for cryptographic applications," in *Proceedings of 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '04)*, vol. 3156 of *Lecture Notes in Computer Science*, pp. 44–61, Cambridge, Mass, USA, August 2004, http://eprint.iacr.org/2004/198/.

[12] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading, Mass, USA, 3rd edition, 1997.

[13] J. Stein, "Computational problems associated with Racah algebra," *Journal of Computational Physics*, vol. 1, no. 3, pp. 397–405, 1967.

[14] R. P. Brent and H. T. Kung, "Systolic VLSI arrays for linear-time GCD computation," in *Proceedings of International Conference on Very Large Scale Integration (VLSI' 83)*, V. Anceau and E. J. Aas, Eds., pp. 145–154, Trondheim, Norway, August 1983.

[15] B. S. Kaliski Jr., "The Montgomery inverse and its applications," *IEEE Transactions on Computers*, vol. 44, no. 8, pp. 1064–1065, 1995.

[16] E. Savaş and Ç. K. Koç, "The Montgomery modular inverse-revisited," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 763–766, 2000.

[17] R. Lórencz, "New algorithm for classical modular inverse," in *Proceedings of 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '02)*, vol. 2523 of *Lecture Notes in Computer Science*, pp. 57–70, Redwood Shores, Calif, USA, August 2002.

[18] T. Jebelean, "Comparing several GCD algorithms," in *Proceedings of 11th IEEE Symposium on Computer Arithmetic (ARITH-11 '93)*, pp. 180–185, Windsor, Ontario, Canada, June-July 1993.

[19] B. Vallée, "Complete Analysis of the Binary GCD Algorithm," 1998, http://citeseer.ist.psu.edu/79809.html.

[20] R. Schroeppel, H. Orman, and S. O'Malley, "Fast key exchange with elliptic curve systems," Tech. Rep. 95-03, Department of

Computer Science, The University of Arizona, Tucson, Ariz, USA, March 1995.

[21] T. Jebelean, "A double-digit Lehmer-Euclid algorithm for finding the GCD of long integers," *Journal of Symbolic Computation*, vol. 19, no. 1–3, pp. 145–157, 1995, Technical report version also available ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1992/92-69.ps.gz.

[22] K. Weber, "The accelerated integer GCD algorithm," *ACM Transactions on Mathematical Software*, vol. 21, no. 1, pp. 111–122, 1995.

Laszlo Hars has earned his doctoral degree at the Eötvös Loránd University, Budapest, in computational geometry. For 15 years, he lectured there in various discrete and numerical mathematics subjects, and computer science. For extended periods, Laszlo visited Kyoto University (working on networked information systems) and the University of Bonn (doing operation research: optimum routing, Steiner trees, traveling salesman type problems). Since 1990, he has been working in industrial research in various fields, like digital signal processing, electric circuit simulation and design, large-scale optimizations, digital watermarking, tamper resistant software, digital rights management, random number generation and testing, and various other areas in information security. Since 2002 he has been working at Seagate Research in Pittsburgh, leading research projects related to storage security. He has published 34 scientific papers and conference presentations, he is the author or coauthor of 7 patents, and over 30 filed patent applications.