GCD-Free Algorithms for Computing Modular Inverses

Marc Joye[1] and Pascal Paillier[2]

[1] Gemplus, Card Security Group
La Vigie, Avenue du Jujubier, ZI Athélia IV, 13705 La Ciotat Cedex, France
marc.joye@gemplus.com – http://www.geocities.com/MarcJoye/
[2] Gemplus, Cryptography Group
34 rue Guynemer, 92447 Issy-les-Moulineaux, France
pascal.paillier@gemplus.com
http://www.gemplus.com/smart/

Abstract. This paper describes new algorithms for computing a modular inverse $e^{-1} \bmod f$ given coprime integers e and f. Contrary to previously reported methods, we neither rely on the extended Euclidean algorithm, nor impose conditions on e or f. The main application of our gcd-free technique is the computation of an RSA private key in both standard and CRT modes based on simple modular arithmetic operations, thus boosting real-life implementations on crypto-accelerated devices.

Keywords: Modular inverses, RSA key generation, prime numbers, efficient implementations, embedded software, GCD algorithms.

1 Introduction

The usual way one computes a modular inverse is by applying the extended Euclidean algorithm [8, Algorithm X, p. 325]. Given e and f on input, this algorithm returns integers α and β such that $\alpha e + \beta f = \gcd(e, f)$. Assuming that e is relatively prime to f, and therefore that e has an inverse modulo f, it follows that $\alpha e \equiv 1 \pmod f$ meaning that $\alpha \equiv e^{-1} \pmod f$.

Unfortunately, for code-optimization reasons, this algorithm is hardly available on embedded platforms. Instead, algebraic tricks based on simple modular arithmetic are highly preferred because gcd-type calculations may be too intricate to handle on cryptoprocessors compared to modular operations. As an example, executing an extended binary gcd may require much less arithmetic or logic operations on large numbers than *glue instructions* such as register switches, loop control, pointer management, etc., rendering this approach comparatively prohibitive to straightforward, arithmetic-only implementations. But then, one requires that one of the two input values, e or f, is prime. Indeed, when f is prime, the inverse of e modulo f is given by Fermat Little Theorem stat-

C.D. Walter et al. (Eds.): CHES 2003, LNCS 2779, pp. 243–253, 2003.
© Springer-Verlag Berlin Heidelberg 2003

ing that $e^{-1} \equiv e^{f-2} \pmod{f}$; when e is prime, $e^{-1} \mod f$ is given by Arazi's well-known inversion formula.[1] Little is known about the other cases.

This paper presents simple ways for computing $e^{-1} \bmod f$ without the (extended) Euclidean algorithm (or variants thereof) and *without any restrictions* on e or f. Our techniques only invoke usual basic operations like (possibly modular) additions, multiplications and exponentiations. Since these operations are optimized on devices supporting public-key cryptography, the technique we propose is especially well-suited for smart-card on-board computation of an RSA private key, in both standard and CRT modes.

2 Arazi's Inversion Formula

When f is prime, the inverse of e modulo f is given by Fermat Little Theorem because $d = e^{-1} \bmod f = e^{f-2} \bmod f$. When f is not prime and *provided that* e *is prime*, the usual trick consists in applying Arazi's inversion formula, which expresses $e^{-1} \bmod f$ in terms of $f^{-1} \bmod e$.

Lemma 1 (Arazi). *Let e and f be two positive integers. If* $\gcd(e, f) = 1$ *then*

$$d = e^{-1} \bmod f = \frac{1 + f(-f^{-1} \bmod e)}{e} \ . \tag{1}$$

Proof. Define $U = e(e^{-1} \bmod f) + f(f^{-1} \bmod e)$. Since $U \equiv 1 \pmod{e}$ and $U \equiv 1 \pmod{f}$, it follows that $U \equiv 1 \pmod{ef}$. Hence, noting that $1 < e + f \leq U < 2ef$, this implies that $U = 1 + ef$, or equivalently that $e^{-1} \bmod f = [1 + f\{e - (f^{-1} \bmod e)\}]/e$ as desired. □

Hence if e is prime, its inverse d modulo f can easily be computed as

$$d = \frac{1 + f(-f^{e-2} \bmod e)}{e} \ .$$

This formula is limited to prime values for e, but is easily extended to

$$d = \frac{1 + f(-f^{\lambda(e)-1} \bmod e)}{e} \ , \tag{2}$$

whenever $\lambda(e)$ is known. We recall that computing Carmichæl's function $\lambda(e)$ from e requires to factor e, a task which imposes a very strong computational requirement. So, the extended technique given by Eq. (2) is of no interest if the inversion algorithm is not given $\lambda(e)$ as an input. The same remarks are independently stated in [6].

[1] Named after Arazi who was the first to take advantage of this folklore theorem to implement fast modular inversions of RSA exponents on a crypto-processor.

2.1 Implementing Arazi's Formula with Modular Operations

Equation (2) requires an integer division (i.e., over $\mathbb{Z}$), which is performed either directly by the cryptoprocessor (some of them may provide this functionality) or in the following way. We compute $d = d \bmod 2^{|f|}$ where $|f|$ stands for the binary length of f. The multiplication and incrementation in Eq. (2) are done modulo $2^{|f|}$, and then the division is replaced by a multiplication by $e^{-1} \bmod 2^{|f|}$. This value may be hard-coded in the program when this is possible, or dynamically computed as depicted on the algorithm of Fig. 1. Another algorithm for performing an integer division can be found in [5, p. 235].

Input: e (odd), $|f|$
Output: $e^{-1} \bmod 2^{|f|}$

$T \leftarrow \lceil \log_2(|f|) \rceil$; $y \leftarrow 1$
for $i = 1$ to T do
  $y \leftarrow y(2 - ey) \bmod 2^i$
endfor
return $y \bmod 2^{|f|}$

Fig. 1. Inversion $e \mapsto e^{-1} \bmod 2^{|f|}$

Note that all operations involved here are modular. Besides, this technique turns out to be extremely fast (only 10 iterations for a typical size $|f| = 1024$), especially for small values of e.

2.2 The Case of Composite Numbers

In the sequel, Π will always denote the product of small primes $\Pi = \prod_{i \in I} p_i$ for $I \subset \mathbb{N}$ and where p_i is the i^{th} prime (i.e., $p_1 = 2$, $p_2 = 3$, ...). Unless stated otherwise, we assume that $I = [1, k]$ for a certain bound k depending on the context of use for Π. We also assume that the choice for Π has been done once and for all, and that Π and $\lambda(\Pi)$ are absolute constants coded in our algorithms.

Now suppose that f is some composite number with unknown factorization. We consider different scenarios depending on the information we have about the operand e:

1. e is known at compile time. We thus have access to $\lambda(e)$ which may be written or coded in the program itself;
2. e is an input data and is provided along with $\lambda(e)$;
3. e is provided alone, but is known to be prime (and thus $\lambda(e) = e - 1$);
4. e is given (e.g., dynamically loaded and provided) but nothing else is known about e.

The first three situations lead to the implementation given below. The fourth context of use is somewhat more intricate and requires a specific treatment as shown later in Section 3.

We adopt the following twofold approach:

- one attempts to retrieve $\lambda(e)$, or a multiple thereof, in order to invoke the previous, very efficient technique;
- if unsuccessful, one computes d without that knowledge but in an heavier, somewhat pathological way.

In some cases, retrieving $\lambda(e)$ may be quite efficient. When e is smooth enough so that $e \mid \Pi$, then a multiple of $\lambda(e)$ is simply $\lambda(\Pi)$. This may also hold without necessarily having $e \mid \Pi$. In many situations, the following will be sufficient. Set

$$\widehat{\lambda} = e(e-1)\lambda(\Pi)\Pi \ , \tag{3}$$

and execute the inversion algorithm of the previous section by replacing $\lambda(e)-1$ by $\widehat{\lambda}-1$. Get the result $\widehat{d}$ and test whether $e\,\widehat{d} \equiv 1 \pmod{f}$. We output $\widehat{d}$ if this equality holds. Otherwise, we know that the structure of e is less simple than originally thought.[2]

The next section describes new efficient approaches that always return the value of $d = e^{-1} \bmod f$ *whatever* the conditions on e and/or f.

3 Extended Algorithms for Composite Integers

3.1 Algorithm 1

Our idea is fairly simple. It is based on the somewhat obvious observation that

$$e^{-1} \equiv (e + Cf)^{-1} \pmod{f}$$

for any integer C. Therefore we can add an appropriate multiple of f to e so that the result is prime and then apply Arazi's inversion formula directly to it. Define

$$\hat{e} = e + Cf \ .$$

We require $\hat{e}$ to be a prime. A naive way to find such an $\hat{e}$ consists in trying $C = 1, 2, \ldots$ and so on until $e+Cf$ is prime [4]. We can however do much better.

Proposition 1. *Let e and f be two positive integers with $\gcd(e, f) = 1$. Let also $\Pi = \prod p_i$ be a product of (small) primes. Define*

$$\hat{e} = e + Cf \quad \textit{with} \quad C = \left[1 - e^{\lambda(\Pi)}\right] \bmod \Pi \ . \tag{4}$$

Then we have $\gcd(\hat{e}, p_i) = 1$ for all primes p_i dividing Π. Moreover, we have $\gcd(\hat{e}, f) = 1$.

[2] In this event, e is a composite number with at least one prime factor e_i with $e_i \nmid \Pi$ such that $e_i - 1$ has some large prime factor ...

Proof. (i) Consider first the case $\gcd(f, p_i) = p_i$. Then from Eq. (4) we have $\hat{e} \equiv e \pmod{p_i}$. Moreover, since by definition $\gcd(e, f) = 1$, it follows that $\gcd(\hat{e}, p_i) = \gcd(e, p_i) = 1$.
(ii) Suppose now that $\gcd(f, p_i) = 1$. If $\gcd(e, p_i) = p_i$ then Eq. (4) yields $C \equiv 1 \pmod{p_i}$, which implies $\hat{e} \equiv f \pmod{p_i}$ and consequently $\gcd(\hat{e}, p_i) = \gcd(f, p_i) = 1$. Conversely, assuming $\gcd(e, p_i) = 1$ forces $C \equiv 0 \pmod{p_i}$ and then $\hat{e} \equiv e \pmod{p_i}$ which, again, leads to $\gcd(\hat{e}, p_i) = \gcd(e, p_i) = 1$.

Finally, as $\hat{e} \equiv e \pmod{f}$, it follows that $\gcd(\hat{e}, f) = \gcd(e, f) = 1$. □

As $\hat{e}$ is co-prime to all small primes $p_i \mid \Pi$, it is likely to be a prime number which we can test using some primality test.[3] If the test is unsuccessful, we re-iterate the process with another candidate $\hat{e}^{(\text{new})} = \hat{e}^{(\text{old})} + f\Pi$, and so forth until $\hat{e}$ is a prime number. Remark that the updated $\hat{e}$, $\hat{e}^{(\text{new})}$, satisfies $\hat{e}^{(\text{new})} \equiv \hat{e}^{(\text{old})} \pmod{\{p_i, f\}}$ and so also verifies Proposition 1.

We note that our technique differs from the one described in [7] in several ways, and in particular, the building of $\hat{e}$ from e is not probabilistic. The resulting algorithm is detailed on Fig. 2.

Input: e, f with $\gcd(e, f) = 1$
$\Pi = \prod p_i$, $\lambda(\Pi)$
Output: $d = e^{-1} \bmod f$

$C \leftarrow \left[1 - e^{\lambda(\Pi)}\right] \pmod{\Pi}$; $\hat{e} \leftarrow e + Cf$
while ($\hat{e}$ is not prime) do
  $\hat{e} \leftarrow \hat{e} + f\Pi$
endwhile
$u \leftarrow f^{\hat{e}-2} \bmod \hat{e}$; $d \leftarrow [1 + f(\hat{e} - u)]/\hat{e}$
return d

Fig. 2. A first inverting algorithm (Algorithm 1)

There may exist variations of Algorithm 1. To illustrate the diversity of our technique, we provide here another alternative. We state:

Proposition 2. *Let e and f be two positive integers with $\gcd(e, f) = 1$. Let also $\Pi = \prod p_i$ be a product of (small) primes, and $c \in \mathbb{Z}_\Pi^*$. Define*

$$\widehat{e} = e + Cf \quad \textit{with} \quad C = \left[(c - e) f^{\lambda(\Pi)-1}\right] \bmod \Pi \ . \tag{5}$$

Then $\gcd(\widehat{e}, p_i) = 1$ for all p_i dividing Π.

[3] Popular primality tests like Fermat's test or Miller-Rabin's test are easy to implement with basic modular operations and hence are quite fast in practice.

Proof. (i) Consider first the case $\gcd(f, p_i) = p_i$. Then from Eq. (5) we have $\widehat{e} \equiv e \pmod{p_i}$. Moreover, since by definition $\gcd(e, f) = 1$, it follows that $\gcd(\hat{e}, p_i) = 1$.
(ii) Suppose now that $\gcd(f, p_i) = 1$. Then Eq. (5) yields $\hat{e} \equiv c \pmod{p_i}$ and thus we find again $\gcd(\widehat{e}, p_i) = 1$ since $c \in \mathbb{Z}^*_{\Pi}$. □

As $\widehat{e}$ is co-prime to all small primes $p_i \mid \Pi$, it is likely a prime number. Otherwise we re-iterate the process with another candidate $c \in \mathbb{Z}^*_{\Pi}$.

Input: f, e, Π, and $a \in \mathbb{Z}^*_{\Pi} \setminus \{1\}$
Output: d_f

---

1. Compute $U \leftarrow f^{\lambda(\Pi)-1} \bmod \Pi$
2. Set $C \leftarrow [(c-e)U \bmod \Pi]$ and $\widehat{e} \leftarrow e + Cf$
3. If ($\mathsf{T}(\widehat{e}) =$ false) then
   a) Set $c \leftarrow a\,c \pmod{\Pi}$
   b) Go to Step 2
4. Compute $F \leftarrow -f^{\widehat{e}-2} \bmod \widehat{e}$
5. Output $d_f = (Ff + 1)/e$

Fig. 3. An alternative algorithm (Algorithm 1')

Again, note that all operations in the above algorithm exclusively rely on basic modular arithmetic. If the cryptoprocessor cannot handle integer divisions directly, the division by e in the last step can be computed with the algorithm described in Fig. 1.

3.2 Algorithm 2

A second algorithm can be derived by exchanging the roles of e and f in Proposition 1. Doing so, we obtain a prime $\hat{f}$. Two applications of Arazi's formula will give thus the expected result.

Since $\hat{f}$ is prime, the inverse of e modulo $\hat{f}$ is given by $u = e^{\hat{f}-2} \bmod \hat{f}$. Noting that $f \equiv \hat{f} \pmod{e}$, a first application of Arazi's formula enables to recover the value of $f^{-1} \bmod e$ as

$$v := f^{-1} \bmod e = \frac{1 + e(\hat{f} - u)}{\hat{f}}\ , \tag{6}$$

and a second application yields

$$d = e^{-1} \bmod f = \frac{1 + f(e - v)}{e}\ .$$

In many cases, the value of e is small compared to that of f. Moreover, using the fact that $\gcd(e, f) = \gcd(e \bmod f, e)$, we obtain the following corollary of Proposition 1.

Corollary 1. *With the notations of Proposition 1, define* $\bar{e} = e \bmod f$ *and*

$$\hat{\bar{e}} = \bar{e} + Cf \quad \textit{with} \quad C = \left[1 - \bar{e}^{\lambda(\Pi)}\right] \bmod \Pi \ .$$

Then we have $\gcd(\hat{\bar{e}}, p_i) = 1$ *for all primes* p_i *dividing* Π*. Moreover, we have* $\gcd(\hat{\bar{e}}, f) = 1$*.*

Proof. Straightforward by replacing e with $\bar{e}$ in Proposition 1. □

Therefore, we can advantageously consider $\hat{f}$ instead of f (remember that for our second algorithm the roles of e and f are exchanged in Proposition 1) and so evaluate v in Eq. (6) as

$$v = \bar{f}^{-1} \bmod e = \frac{1 + e(\hat{\bar{f}} - u)}{\hat{\bar{f}}}$$

where $\bar{f} = f \bmod e$.

Putting all together, we obtain a second algorithm for computing modular inverses.

Input: e, f with $\gcd(e, f) = 1$
$\Pi = \prod p_i$, $\lambda(\Pi)$
Output: $d = e^{-1} \bmod f$

$\hat{\bar{f}} \leftarrow f \bmod e$
if ($\hat{\bar{f}}$ is not prime) then
  $\bar{C} \leftarrow \left[1 - \hat{\bar{f}}^{\lambda(\Pi)}\right] \pmod{\Pi}$; $\hat{\bar{f}} \leftarrow \hat{\bar{f}} + \bar{C}e$
  while ($\hat{\bar{f}}$ is not prime) do
    $\hat{\bar{f}} \leftarrow \hat{\bar{f}} + e\Pi$ [*i*]
  endwhile
endif
$u \leftarrow e^{\hat{\bar{f}}-2} \pmod{\hat{\bar{f}}}$
$d \leftarrow [\hat{\bar{f}} + f(e\,u - 1)]/(e\hat{\bar{f}})$ [*ii*]
return d

Fig. 4. Our second algorithm (Algorithm 2)

This second algorithm is particularly efficient when e is small since then Π may be chosen smaller, which in turn implies smaller values for $\bar{C}$ and for $\hat{\bar{f}}$. On

the contrary, the first algorithm (Fig. 2) is more suitable when the size of e is sensibly the same as that of f.

As easily seen, the choice for parameter Π remains completely free. We now discuss the best way to choose Π in practice. Primality checks executed in our 'while' loop involve integers of bitsize close to $|e\Pi|$. As most practical implementations for primality testing are of cubic complexity, a single test has a cost $\propto |e\Pi|^3$. Moreover, the average number of tests amounts to

$$|e\Pi| \cdot \ln 2 \cdot \frac{\phi(\mathrm{lcm}(e, \Pi))}{\mathrm{lcm}(e, \Pi)} \ .$$

Totalling these two facts, and upper bounding the ratio $\phi(\mathrm{lcm}(e, \Pi))/\mathrm{lcm}(e, \Pi)$ by $\phi(\Pi)/\Pi$, the average workfactor for finding $\hat{\hat{f}}$ is bounded by a function proportional to $(|e\Pi|)^4\phi(\Pi)/\Pi$. Therefore, provided that $\Pi = \prod_1^k p_i$, an optimal choice for k with respect to a given $|e|$ is easily found. Interestingly, for small parameter lengths such as $|e| = 32$ or 64, the optimum is obtained for $k = 3$ ($\Pi = 2 \cdot 3 \cdot 5$), i.e., for an extremely small value of Π. Algorithm 2 then performs only a few primality checks over integers of size close to the one of e, and is therefore very fast.

Remark 1. Certain hardware implementations return the value of $f \operatorname{div} e$ together with the value of $f \bmod e$ when computing the remainder of an integer division. In this case, the division by $e\hat{\hat{f}}$ in the expression of d (cf. [‡] in Fig. 3) can be reduced to a division by $\hat{\hat{f}}$. Initializing $\bar{C}$ to 0 and keeping track of its accumulated value, we can replace Line $[i]$ by

$$\bar{C} \leftarrow \bar{C} + \Pi; \hat{\hat{f}} \leftarrow \hat{\hat{f}} + e\Pi \qquad [i']$$

and Line $[ii]$ by

$$d \leftarrow [fu - (f \operatorname{div} e) + \bar{C}]/\hat{\hat{f}} \ . \qquad [ii']$$

4 Application to RSA

RSA [11], named after its inventors Rivest, Shamir, and Adleman, is undoubtedly the most widely used cryptosystem. We give hereafter a short description and refer the reader to the original paper or any textbook on cryptography for further details.

Let $n = pq$ be the product of two large primes. We let e and d denote a matching pair of public exponent/private exponent, according to

$$e\,d \equiv 1 \pmod{\lambda(n)} \ , \qquad (7)$$

where λ is Carmichæl function. In particular, for the RSA, we have $n = pq$ and $\lambda(n) = \mathrm{lcm}(p - 1, q - 1)$.

Given $x \in]0, n[$, the public operation (e.g., encryption of a message or verification of a signature) consists in raising x to the power e, modulo n, i.e., in computing $c = x^e \bmod n$. Next, from c, the corresponding private operation (e.g., decryption of a ciphertext or a signature generation) is $c^d \bmod n$. From the definition of e and d, we obviously have that $c^d \equiv x \pmod{n}$.

4.1 Standard Mode

In standard mode, on input p, q and e, one has to compute the private exponent d satisfying Eq. (7). We assume that we are given Π, the product of small primes, along with $\lambda(\Pi)$. These numbers are pure constants, and are thus easily hard-coded into the implementation.

When e (or its factorization) cannot be determined in advance, a direct application of Algorithm 1 (or Algorithm 2) with e and $f = \mathrm{lcm}(p-1, q-1)$ on input will output the corresponding secret key d.

If the value of $\mathrm{lcm}(p-1, q-1)$ cannot be computed, one can replace the Carmichæl function of n by the Euler totient function and take $f = (p-1)(q-1)$. This, however, results in a larger yet valid value for d.

From a computational viewpoint, taking $|f| = 1024$ and $|e| = 32$ for instance, a typical implementation of Algorithm 2 would use the specific choice $\Pi = 2 \cdot 3 \cdot 5$. Thus, around 6.83 primality tests over 37-bit numbers are required, on average. When $|f| = 1024$ and $e = 64$, the same choice for Π yields an average of 12.75 primality tests over 69-bit numbers. In addition to that, operations starting and ending the algorithm are almost negligible: computing $\bar{C}$ amounts to a few squares modulo $2 \cdot 3 \cdot 5$; u requires an exponentiation of size close to $|e|$; and the computation of d (thanks to our technique on Fig. 1) boils down to a few multiplications carried out modulo 2^{1024}.

4.2 CRT Mode

The private operation can be speeded up through Chinese remaindering (CRT mode) [9]. The computations are performed modulo p and q and then recombined. The private parameters are (p, q, d_p, d_q, i_q) with $d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$ and $i_q = q^{-1} \bmod p$. We then obtain $c^d \bmod n$ as

$$\mathrm{CRT}(x_p, x_q) = x_q + q[i_q(x_p - x_q) \bmod p] \ ,$$

where $x_p = c^{d_p} \bmod p$ and $x_q = c^{d_q} \bmod q$. The expected speed-up factor is 4, compared to the standard (i.e., non-CRT) mode.

In CRT mode, the procedure is readily the same. We apply Algorithm 1 or 2 where inputs e and f are initialized to $e \pmod{(p-1)}$ and $p-1$, respectively. This yields the value of the private exponent d_p. Similarly, the exponent d_q is obtained by initializing e and f to $e \pmod{(q-1)}$ and $q-1$, respectively.

4.3 Standard Mode (II)

There is another way to compute the private key in standard mode. We first compute d_p and d_q as described in the previous section. Next, letting $\mathcal{Q} := q-1$ and $\Lambda := \lambda(n)/(q-1)$, we compute the inverse of $\mathcal{Q}$ modulo Λ, say $\mathcal{I}_{\mathcal{Q}}$, thanks to the algorithm of Fig. 2 as[4]

[4] Remark here that we have to compute the inverse of $(q-1)$ modulo $\frac{p-1}{\gcd(p-1,q-1)} = \frac{\mathrm{lcm}(p-1,q-1)}{q-1}$ as $e^{-1} \bmod f$ exists if and only if $\gcd(e, f) = 1$.

$$\mathcal{I}_{\mathcal{Q}} = \frac{1 + \Lambda \left[-\Lambda^{\widehat{\mathcal{Q}}-2} \bmod \widehat{\mathcal{Q}} \right]}{\widehat{\mathcal{Q}}} \tag{8}$$

where

$$\widehat{\mathcal{Q}} = q - 1 + C_{\mathcal{Q}}\, \Lambda \quad \text{with} \quad C_{\mathcal{Q}} = \left[1 - (q-1)^{\lambda(\Pi)} \right] \bmod \Pi + \mu \cdot \Pi$$

for some $\mu \geq 0$ such that $\widehat{\mathcal{Q}}$ is prime. Therefore, Chinese remaindering on d_p and d_q finally gives $d = d_q + (q-1)\,[\mathcal{I}_{\mathcal{Q}}(d_p - d_q) \bmod \Lambda]$.

5 Conclusion

We devised new algorithms for computing modular inverses in a gcd-free manner. We stress that, implementing our techniques, an RSA key generation process can be executed on any given crypto-enhanced embedded processor in almost every circumstances.

Acknowledgements. We are grateful to Jean-François Dhem for pointing out reference [4] and to Karine Villegas for her careful reading of an earlier version of this paper.

References

1. A.O.L. Atkin and F. Morain. Elliptic curves and primality proving. *Mathematics of Computation* **61**:29–68, 1993.
2. D. Boneh and M. Franklin. Efficient generation of shared RSA keys. In *Advances in Cryptology – CRYPTO '97*, vol. 1294 of Lecture Notes in Computer Science, pp. 425–439, Springer-Verlag, 1997.
3. W. Bosma and M.-P. van der Hulst. Faster primality testing. In *Advances in Cryptology – CRYPTO '89*, vol. 435 of Lecture Notes in Computer Science, pp. 652–656, Springer-Verlag, 1990.
4. M.F.A. Derôme. Generating RSA keys without the Euclid algorithm. *Electronics Letters* **29**(1):19–21, 1993.
5. S.R. Dussé and B.S. Kaliski Jr. A cryptographic library for the Motorola DSP 56000. In *Advances in Cryptology – EUROCRYPT '90*, vol. 473 of *Lecture Notes in Computer Science*, pp. 230–234, Springer-Verlag, 1991.
6. W. Fischer and J.-P. Seifert. Note on fast computation of secret RSA exponents. In *Information Security and Privacy (ACISP 2002)*, vol. 2384 of *Lecture Notes in Computer Science*, pp. 136–143, Springer-Verlag, 2002.
7. M. Joye and P. Paillier. Constructive methods for the generation of prime numbers. *Proc. of the 2nd NESSIE Workshop*, Egham, UK, September 12–13, 2001.
8. D.E. Knuth. *The Art of Computer Programming, v. 2. Seminumerical Algorithms.* Addison-Wesley, 2nd edition, 1981.
9. J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters* **18**:905–907, 1982.
10. H. Riesel. *Prime Numbers and Computer Methods for Factorization*, Birkhäuser, 1985.

11. R.L. Rivest, Adi Shamir, and L.M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* **21**(2):120–126, 1978.
12. R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing* **6**: 84–85, 1977.