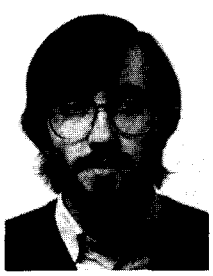
Implementing the RSA Cryptosystem

Achim Jung
Technische Hochschule Darmstadt, Fb. 4, 6100 Darmstadt, West Germany

Techniques for a software implementation of the RSA cryptosystem are presented. They allow both space and time requirements of the RSA scheme to be minimized. A hash function to be used in connection with the cryptosystem is presented; this function avoids weaknesses of other hash functions published previously. Performance results of several implementations are given, which show that the RSA algorithm is acceptably fast for a large number of applications.

Keywords: Cryptology, RSA cryptosystem, Digital signatures, Primality tests, Hash functions



Achim Jung studied mathematics and computer science at the Technical University in Darmstadt in West Germany. After graduation, he worked for the German Research Institute for Mathematics and Computer Science where most of the ideas presented in this paper were developed. In summer 1984 he entered Carnegie-Mellon University in Pittsburgh, USA.
Mr. Jung is currently an assistant in the algebra working group of the Mathematics Department at the Technical University of Darmstadt where he is preparing for his Ph.D. In addition to cryptology, he is interested in the mathematical foundations of computer science and mathematical and general educational sciences.

This work was done under contract for the GMD as part of the OSIS project. It was partly sponsored by the European Commission under COST 11bis and by the German Ministry of Research and Technology.

North-Holland
Computers & Security 6 (1987) 342–350

1. Introduction

Nearly ten years have passed since R.L. Rivest, A. Shamir and L. Adleman [26] gave the first realization of a public-key cryptosystem ingeniously designed by Diffie and Hellman in 1975. After early enthusiastic reactions people were putting their hope in the development of an RSA chip, which would effectively perform the necessary time consuming multiple precision routines. Then as now the opinion was prevalent that the RSA algorithm could not be implemented effectively in software. When the 'Gesellschaft für Mathematik und Datenverarbeitung (GMD)' in Darmstadt, West Germany, initiated the OSIS project three years ago, the original idea was to simplify the access to electronic information services by paying immediately via electronic 'checks' (see [31] for detailed information on OSIS). We did not know ourselves whether a software implementation would ever be practicable.

The main purpose of this paper is to show that it can be done effectively, to describe how we did it and thus to encourage others to use this infinitely flexible instrument right now. Even when RSA chips are easily available (there are several developments ongoing in the US and Europe), there will still be the need to have the routines on the existing mainframe computers as well. It is also our intention to propose and popularize specific solutions on how to optimally apply and implement the RSA method in order to foster a consensus among designers, which would help to develop standards in this area.

We could not prove the correctness – in the appropriate sense – of all of our suggestions exhibited below. We therefore strongly invite comments on them. This applies especially to the 'unbalanced RSA scheme' (Sec. 2.2), the considerations about primality testing (Sec. 3), and our hash function (Sec. 4).

2. Optimization of the RSA Cryptosystem

In this section we show how the time and space requirements of the original RSA algorithm can be

minimized. In what follows we adopt the notation of the original RSA publications [26,24], that is: $n = p \cdot q$ denotes the modulus, p', q' are prime factors of $p-1$, $q-1$, and p'', q'' are prime factors of $p'-1$, $q'-1$, respectively, e is the encryption exponent, d is the decryption exponent, and m denotes a message, viewed as a number. We will not repeat the basics of asymmetric cryptosystems nor of the RSA scheme, since it seems to us that it is nearly impossible to improve on the original exposition in [26].

2.1 Common Encryption Exponent

We let every participant share the same encryption exponent, which we chose to be the fifth Fermat prime $65\,537 = 2^{16}+1$, and thus define the modulus n as the user-specific public key. The advantages of this convention are twofold: encryption/authentication takes only 17 multiplications and is thus about 30 times faster than decryption. It is not necessary to store the encryption exponents of other users; we save memory space and transmission overhead.

Taking $e = 3$, as Knuth ([13], p. 386ff) suggests, is not advisable. A common situation in a network will be that the same secret message m has to be sent to different recipients. If the encryption exponent is equal to 3 and an intruder succeeds in recording the encrypted messages c_1, c_2, c_3, sent to three different participants, he can decipher the message m. He proceeds as follows:
Since m^3 is the solution to the simultaneous linear congruences

$$x \equiv c_1 \bmod n_1$$
$$x \equiv c_2 \bmod n_2$$
$$x \equiv c_3 \bmod n_3$$

and since there is only one solution between 0 and $n_1 \cdot n_2 \cdot n_3 - 1$, m^3 can be calculated by the Chinese Remainder Theorem. Extracting cube roots can be done effectively and so m is revealed. (We would like to note here that sharing the same modulus is not secure as was shown in [4] and [27].)

2.2 The Unbalanced RSA Scheme

Still more memory space can be saved, if all moduli are very close to a power of 2. By very close we mean that the inequality

$$n - 2^{\alpha} < 2^{\beta}$$

is satisfied for some $\beta < \alpha$. If in some application a fixed α is chosen, then only the difference $n - 2^{\alpha}$, which has only β binary digits, needs to be stored or transmitted. The idea of using only such special moduli is due to C.P. Schnorr, who could guarantee this form with $\beta = \alpha/2$ easily in his OSS scheme [16] (The OSS scheme, which was designed to be much faster than RSA, was broken by J.M. Pollard [18], unfortunately.) We show how to get these moduli for the RSA scheme. This is more difficult because of the restrictions imposed on the factors p and q (cf. [24]).

We start with generating primes q'', q', q, p'', p' in the usual way, except that they are of size approximately $2^{\beta/2}$. The second factor p of n is then chosen to be the smallest prime such that p' divides $p-1$ and $p \cdot q$ is greater than 2^{α}. This implies that p will be of size $2^{\alpha-\beta/2}$. This special 'unbalanced' choice of the factors might simplify factoring the modulus, but we do not know to what extent. Especially Pollard's ρ-method [17], which searches for the smallest factor, has to be considered here. With this factorization algorithm existing, choosing $\beta = \alpha/2$ seems to be risky. But $\beta = 0.8 \cdot \alpha$ (Actually this depends on α and the maximal size of an integer value in the machine used. We assume $\alpha = 512$ and a 32 bit computer word here.) will already guarantee that the modulus starts with digits '1000...', where there are at least 63 zero bits following the leading 1. This will greatly simplify the 'mod' operation, since no trial division and no check, as to whether the trial digit exceeds the true digit by 2 is needed (Step D3 of Knuth's 'Algorithm D', [13], p. 257f). Also, it will happen only very rarely that the trial digit is one too large. Thus Step D6 of the algorithm will be executed with low probability only.

2.3 Speeding up Decryption

Quisquater and Couvreur [22] have published a method for speeding up decryption/signing by a factor of 4. The idea is quite simple and will be repeated here: Since every user has to keep his private key d secret, he might as well store the factors p and q together with $dp := d \bmod p-1$ and $dq := d \bmod q-1$ instead of n and d. (only a little more memory space is needed for that.)

Exponentiation is then done modulo p and q, respectively, and the solution with respect to n is calculated via the Chinese Remainder Theorem. Since exponentiation is of cubic complexity, we get a speed-up by a factor of 4. If a common encryption exponent protocol is adopted, storing dp and dq is superfluous. For this case we give an algorithmic description of Quisquater and Couvreur's method:

Step 1. Calculate $dp := e^{-1} \bmod p-1$
$dq := e^{-1} \bmod q-1$
$u := = q^{-1} \bmod p$ via the extended Euclidean algorithm (Knuth's 'Algorithm X', [13], p. 325).

Step 2. Calculate $mp := (c \bmod p)^{dp} \bmod p$
$mq := (c \bmod q)^{dq} \bmod q$.

Step 3. Calculate $v := u \cdot (mp - mq) \bmod p$

Step 4. Return $m := mq + q \cdot v$.

Of course, nothing can be done for encryption/authentication in this manner, but since the common encryption exponent is to small, a speed-up is not necessary there.

2.4 Hierarchical Key Structure

Working with a public-key cryptosystem a user needs to have access to a public directory, where all the necessary information about other users' keys is maintained. It was noted very early in the literature that this can be avoided, if a hierarchical key system is adopted. In the following we want to point out a weakness of this protocol, but let us first look at its advantages.

In an hierarchical system every user sends his public key together with his signatures. In order to guarantee authenticity, he also adds the signature of a 'Certification Authority (CA)' under his public key (which is in fact his modulus, since the encryption exponents are all the same). All information a user has to store in order to fully participate in RSA applications then is – all improvements of this section included – the following:

p and q: α bit (secret)

$n - 2^{\alpha}$ (the modulus of the CA): β bit (public)

The CA's signature under the user's modulus:

α bit (public).

This protocol has weaknesses if the *CA* signs the users' moduli by simply raising them to its decryption exponent's power. In this situation it can be attacked as follows:

Take a random number s and pretend that it is the CA's signature under a true user's modulus n. This modulus can be calculated from s by raising it to the CA's encryption exponent's power. There is a fair chance that the resulting n can be factored in a reasonable amount of time. Having the factorization, you can determine the inverse of the common encryption exponent with respect to $\phi(n)$, Euler's totient function. Thus you can produce an imaginary user, whose modulus is signed by the CA and therefore is trusted by other participants.

This attack can be averted in one of the following ways:

- If the unbalanced RSA scheme is used, it is unlikely that a random s will yield a modulus, which is very close to a power of 2. This could make other users suspicious about your modulus. Actually, there would have to be a check in the authentication routine, which validates the relation $N - 2^{\alpha} < 2^{\beta}$.
- If the modulus of the CA is much larger than users' moduli, which might be a desirable feature anyway, then again it is hard to find numbers s, which will result in a modulus of the right size.
- The best way to prevent this kind of attack is to never apply the signing function directly to a submitted text. Instead, every message should first be fed into a 'hash function h' which has the property of being practically noninvertible. (The hash function will also be used to compress long texts to a number less than the actual modulus, so that a single application of the RSA algorithm suffices in the signature routine. More about hash functions in Section 4.) The attack described above breaks down: Starting with a random s, the intruder can easily calculate the corresponding value $h(n)$, but he has no chance of finding the modulus itself. We adopted this protocol in our implementation in that we built the hash function right into the RSA signature routine.

Apart from that, it might be helpful if the CA did sign not just the users's moduli, but rather strings consisting of the user's name, his network address, and his modulus together with an expiration date. Having the hash function at hand, all this can be done in an elegant way.

3. Testing Primality

The most time consuming part of the entire RSA scheme is the generation of keys. Also the most time can be saved here as we shall see. The difficulty comes from the fact that we have to find six large prime numbers p'', p', p, q'', q', q, where p'', (q'') is a factor of $p'-1$ ($q'-1$) and p' (q') is a factor of $p-1$ ($q-1$). (If stronger restrictions are imposed on p and q in order to prevent attacks with certain factorization algorithms – suggestions in this direction were made by the authors of [30] and [9] – even more prime numbers have to be found for a single pair of RSA keys.)

To find such large primes only probabilistic primality tests are feasible. In their original publication, Rivest, Shamir and Adleman suggested that the probabilistic primality test of Solovay and Strassen [29] to ensure primality of a candidate x be applied 100 times. If x passes all tests, the probability that x is still composite is then less than 2^{-100}. Two comments are in order here: First, there are better tests available now, like the one of Rabin [23] called 'Algorithm P' by Knuth ([13], p. 379), which will give the answer 'prime' although x is composite, with probability at most $1/4$. Second, it seems that this is the wrong way to estimate the probability of making a mistake. What Solovay and Strassen proved was that given any composite number x, *at least* 50% of the numbers 2 through $x-1$ are 'witnesses of the nonprimality' of x. But we would like to know how many witnesses there are *on the average*.

This question has not yet been treated in the literature with respect to Solovay and Strassen's or Rabin's test, but the results, which were obtained with respect to the simple Fermat test ($2^{x-1} \equiv 1 \bmod x$) indicate that the fraction of numbers which are witnesses for nonprimality, is very close to 100% on the average. We cite a theoretical and an empirical result to support this point of view. There the question was treated, how many numbers there are, for which 2 is not a witness of their nonprimality. (Such numbers are called 'pseudoprimes.') Assuming that there is nothing special about 2, that is, that the witnesses are evenly distributed over the interval $[2, x-1]$, this will show that on the average nearly every number below x is a witness.

- Upper bounds for the number of pseudoprimes below a certain number were proved by Erdös [7] and by Pomerance [20]. Unfortunately, these bounds will hold for sufficiently large numbers only, but they also show that the number of pseudoprimes not only relative to all numbers, but also relative to the number of primes, converges toward 0.
- Experimental results (cf. [21]) show that the number of pseudoprimes in an interval of constant length is steadily decreasing. The authors of [21] report that they did not find any pseudoprime between 10^{15} and $10^{15}+10^7$. On the other hand there are about 200,000 primes in this interval.

We conclude that having a large arbitrary number x satisfying Fermat's little theorem (or passing Rabin's test with base 2), we can be almost sure that x is indeed prime. In that we agree with the authors of [21].

Even more time can be saved, if we apply Rabin's test only to those numbers which are not divisible by the first m small primes. This was suggested by several authors working in the field (cf. [13,25,11]), but they always gave only vague indications of how big m should be. In the following we give an easy to implement method to find the optimal m. We analyse the following algorithm:

Step 1. Start with a random odd number x.

Step 2. Calculate and store all remainders $r_i := x \bmod p_i$, $1 \leqslant i \leqslant m$. ($p_i$ is the i-th odd prime number.)

Step 3. Set $i \leftarrow 1$, $s \leftarrow 0$.
($x+s$ will be the next prime after x.)

Step 4. If $s + r_i \bmod p_i$ equal 0, increase s by 2 and set $i \leftarrow 1$. Return to Step 4.
Otherwise increase i by 1. If i is greater than m, go to Step 5, otherwise repeat Step 4.

Step 5. Check whether $x+s$ passes Rabin's test. If it does, terminate the algorithm and return $x+s$.
If not, increase s by 2, set $i \leftarrow 1$, and return to Step 4.

In order to calculate the optimal value for m, we first have to determine the time each step in the above algorithm consumes. This can be done with any preferred measure of complexity. We simply chose the number of multiplications/divisions, which has the advantage that a simple relationship between the size of x and the com-

plexity of each step can be established. In the following, we denote by

D_x: the time, which is needed to calculate one remainder x mod p_i in Step 2, by

T: the time, which is needed to test whether $(s + r_i)$ mod p_i equals 0 in Step 4 and by

R_x: the time, which is needed for one application of Rabin's test in Step 5.

We can now express the time S_x, the whole algorithm will need on the average to find the next prime number after a given x in terms of m, D_x, T, and R_x. The time needed for Step 2 equals $m \cdot D_x$. The time needed for Steps 4 and 5 depends on how many numbers $x + s$ we have to test before we find a prime number. The 'Prime Number Theorem' tells us that the density of primes around x amounts to $1/\ln(x)$ in the average and since we are searching through odd numbers only, we have to check about $\ln(x)/2$ numbers on the average.

All candidates are tested to determine if they are divisible by 3 and one third is cancelled out by this step; only two thirds are tested for divisibility by 5, and so on. Rabin's test finally is applied only, if all tests of Step 4 were positive. This happens in $(1 - 1/3) \cdot (1 - 1/5) \cdot \cdots \cdot (1 - 1/p_m)$ of all cases. So we get for S_x:

$$S_x = m \cdot D_x + \frac{\ln(x)}{2} \cdot \left(T + \tfrac{2}{3}T + \cdots + \prod_{i=1}^{m-1} \left(1 - \frac{1}{p_i}\right) \cdot T + \prod_{i=1}^{m} \left(1 - \frac{1}{p_i}\right) \cdot R_x \right)$$

$$= m \cdot D_x + \frac{\ln(x)}{2} \cdot T \cdot \sum_{j=1}^{m} \prod_{i=1}^{j-1} \left(1 - \frac{1}{p_i}\right) + \frac{\ln(x)}{2} \cdot R_x \cdot \prod_{i=1}^{m} \left(1 - \frac{1}{p_i}\right).$$

We observe that increasing m by one will increase the first two terms by

$$D_x + \frac{\ln(x)}{2} \cdot T \cdot \prod_{i=1}^{m} \left(1 - \frac{1}{p_i}\right),$$

and decrease the third term by

$$\frac{\ln(x)}{2} \cdot R_x \cdot \prod_{i=1}^{m} \left(1 - \frac{1}{p_i}\right) \cdot \frac{1}{p_{m+1}}.$$

With growing values of m the gain in the third term becomes less and less, but the loss through the first two terms will always be bigger than D_x. For some m^* gain and loss will cancel each other out and this is exactly the optimal number of small primes we are looking for. So we have to solve the problem

$$\frac{\ln(x)}{2} \cdot R_x \cdot \prod_{i=1}^{m} \left(1 - \frac{1}{p_i}\right) \cdot \frac{1}{p_{m+1}} = D_x + \frac{\ln(x)}{2} \cdot T \cdot \prod_{i=1}^{m} \left(1 - \frac{1}{p_i}\right).$$

To do this effectively, we replace via Merten's Theorem (cf. [10]) the product

$$\prod_{i=1}^{m} \left(1 - \frac{1}{p_i}\right) \text{ by } \frac{2 \cdot e^{-c}}{\ln(p_m)},$$

where c is Euler's constant, and let p range over all real numbers. Thus we have to find the root of the function

$$F(p) := p - \frac{R_x}{D_x \cdot \ln(p)/\ln(x) \cdot e^{-c} + T}.$$

We approximate the root with Newton's method, since it cannot be calculated directly. It is easy to show that the inequalities

$F(p) < p$,

$F'(p) > 1$, and

$F''(p) < 0$

hold for all positive values of p, from which we can infer that Newton's method will converge toward the unique root p^* of $F(p)$ for an arbitrary starting value $p_0 \in \mathbb{R}^+$. After about 5 iterations $|p_i - p^*|$ is less than 10^{-2}. Having the optimal p^*, which is a real number, we look through our table of small primes and search for the nearest prime p_{m^*} to p^*.

In practice, one would not let m^* become arbitrarily big, since storing the primes and the residues of x needs a lot of space. In our implementation we used only primes which fit into a 16 bit word that is the 3511 primes from 3 to 32 749.

(Remark: In order to calculate $\ln(x)$ in the above procedure, one does not need a multiple precision logarithm subroutine. It suffices to calculate the logarithm for the leading digit of x – just as tables for the logarithm were used in older times.)

4. Hash Functions

As noted above, it is desirable to have a secure one-way function h –called 'hash function' from now on – together with the RSA package in order to improve the original algorithm. For convenience we summarize the two basic applications for such a function.

- It should allow one to sign a message of arbitrary length with just one application of the cryptoalgorithm. This is the 'compression feature' of h.
- It should be applied to any message which is submitted for signature. Because of it's 'one-way character' it will be impossible for a cryptoanalyst to forge signatures in a 'plaintext/ciphertext' attack. (Some weaknesses of the RSA cryptosystem in this direction were pointed out by D.E. Denning in [5].) It also allows for a secure hierarchical key management as described in Section 2.4.

From these applications we can draw some conclusions as to what properties the hash function should have.

a. It should be effectively computable and thus yield an advantage over breaking down long messages and signing each piece separately.
b. It should destroy the multiplicative structure underlying the *RSA* scheme. (The signature under a product is the product of the signatures under each factor.)
c. It should be computed over the entire message and should react sensitively to any alterations of the message such as inserting or deleting characters or changing the order of the characters.
d. Since every message is mapped into the interval $[1, n-1]$ by the hash function, infinitely many messages are mapped onto the same number. This implies that it must be infeasible for a cryptoanalyst to find other messages resulting in the same value as a given message. We would like to distinguish two levels of security here:

1. Some other pre-images can probably be found, but almost surely they are of no use for the attacker.
2. Other pre-images cannot be found in a systematic way apart from trying out.

Several hash functions have been proposed: Davies and Price developed one on the basis of the NBS Data Encryption Standard. We decided against such a solution because it would have required additional hard- or software. But of course, this might be an alternative to be considered again in a hybrid DES/RSA cryptosystem. (Many people believe that such a combination will be profitable for both schemes anyway.)

In [12] a hash function is proposed which uses only modular multiplication and modular addition but fails to be secure with respect to requirement d. Since our hash function is based on the same ideas, we will describe this function. The authors of [12] divide a message into blocks X_1 through X_m and calculate recursively

$$Z_0 := c \text{ (some starting value)}$$

$$Z_{i+1} := (Z_i + X_{i+1})^2 \bmod n,\ i = 1, \ldots, m-1.$$

The result is Z_m. Of course, if someone alters X_i, he immediately knows how to alter X_{i+1} or any X_{i+j}, such that the result remains the same. An infinity of pre-images to a given hashed message can be found. (To be fair, it should be noted that Jueneman et al. were dealing with a somewhat different situation where the starting value c could be kept secret.) Davies [3] improves on this by dividing the message into more and shorter blocks, padding each one out with zeros on the left and using the exclusive-or-function instead of ordinary addition. Changing one block X_i will most likely change some of the higher bits of Z_i. These changes cannot be neutralized by changing X_{i+1}, since the higher bits of X_{i+1} are always equal to zero.

We see that adding redundancy to the message is an excellent way to make the hash function better with respect to requirement d, but Davies' scheme still contains a slight weakness. By applying the extended Euclidean algorithm to $Z_{i-1} + X_i$ and n, changes of X_i can be found, which will to alter the higher bits of Z_i, thus allowing an altered X_{i+1} to cancel these changes out. This is possible only because we know that the resulting change in Z_i must be small in order not to affect the higher bits of Z_i. We repair this weakness by one last modification, which distributes redundancy over the message in a mathematically hard to describe fashion. For the sake of clearness and reference we give a quite formal specification of our hash function.

Assume that we are given the user's modulus n as a number between $2^{16 \cdot M}$ and $2^{16 \cdot (M+1)} - 1$ and

a message consisting of k bytes. We proceed as follows:

Step 1. (Filling up) Pad out the message at the end with bytes from the beginning so that you get a string consisting of $k' \cdot M$ bytes.

Step 2. (Zoning) Divide the message into k' blocks X_i^0, each consisting of M bytes. Interpret each blocks as a sequence of half bytes $b_1 b_2 b_3 \ldots b_{2M}$. Precede each half byte with four binary ones, thus producing a string X_i of $2M$ bytes $Fb_1 Fb_2 Fb_3 \ldots Fb_{2M}$.

Step 3. (Compressing) Calculate recursively

$$Z_1 := X_1^2 \bmod n$$

$$Z_{i+1} := (Z_i + X_i)^2 \bmod n,\ i = 1, \ldots, k' - 1.$$

(This is essentially Jueneman et al.'s compression function.)

Step 4. (Normalizing) If $Z_{k'}$ is smaller than $[n/2]$ then add $[n/2]$ to $Z_{k'}$. Return $Z_{k'}$.

The major advantage over Davies' scheme [3] is that there is no systematic way anymore to find changes in X_i, which will leave every other half byte of Z_{i+1} unchanged. Also, using modular addition instead of the exclusive-or-function in Step 3 saves us one extra subroutine.

Objections to our function could be that it is not clear what would happen if the message string was the empty string and that the strings 'a', 'aa',..., '$aa \ldots a$' $= a^M$ will yield the same result because of Step 1. It is doubtful whether this could ever play any role in a signature/authentication application, but there is an easy to implement remedy for it. The first byte of every message is taken to be k mod 256 (k is the number of bytes in the message). Empty strings no longer occur and Step 1 will extend 'a' to '$1a1a1a \ldots$' and 'aa' to '$2aa2aa \ldots$' and so forth.

Step 4 was added, because several public key signature schemes (RSA does not belong to them) are not secure if applied to small numbers. Since we were experimenting with these schemes also, we thought it best to include this normalisation uniformly. The hash function should not depend on the signature scheme used with it.

5. Random Number Generators

Although the original RSA publication was 'algorithmic' in a convenient way, one question was left open, the problem of finding large random numbers. We have to admit that so far we can not solve it in a satisfactory way ourselves. But a recent publication [11] has convinced us that it is necessary to point out again how it should *not* be done.

In the process of key generation a large random number is needed, which serves as a starting point for the search for a prime number. This random number must not be easy to guess because the generated keys depend deterministically on this number. From that it should be clear that it does not suffice to concatenate the output of any pseudorandom number generator, as was done in [11] and some other implementations we know of. The sequence a pseudorandom number generator produces depends deterministically on its first element and so there will be only as many different keys as there are 'first elements.' Knowing this, a cryptoanalyst can search through this relatively small set and find the factorizations of all moduli produced this way.

A moment's thought should convince the reader that there cannot be an algorithmic solution to this problem. John von Neumann (as cited in [13], p. 1) put it into words: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." It is the world of physics, where true randomness – at least from a human point of view – occurs: radioactive decay, Brownian motion, white noise and many other phenomena could be mentioned here. Knuth ([13], p. 387) suggests in connection with his 'RSA-box': "... some scheme based on truly random phenomena in nature like cosmic rays."

Recently a chip was presented to the public (cf. [8]), which uses the frequency instability of an oscillator to produce long random numbers as we need them in our context. Until such chips are widely available, we have to work with intermediary solutions, two of which we now describe.

- One can measure the time, which elapses until the operating system serves some request, such as storage allocation. On a large mainframe system this time is variable enough so the process can be repeated until the required number of 'random' bits is produced. Note that we are implicitly using the 'random behaviour' of other human users.
- Another solution was communicated to us by C.P. Schnorr. He suggested building a

multiple-precision standard pseudorandom number generator and to either keep the original seed or parts of the underlying algorithm secret. We decided against such an approach because it would contradict the philosophy of RSA as a 'public system' and would have put too much responsibility into the key generation center, making it comparable to a governmental agency.

6. Performance of the GMD Implementations

During the last two years we implemented the RSA algorithm on several machines. Assembler subroutines on one hand and the enhancements of the original scheme as exhibited in this article on the other hand resulted in very satisfactory implementations. In fact, many people, who had doubted the practicability of public-key cryptosystems because of a feeling that they are generally slow, could be convinced to develop marketable products based on the RSA scheme. Among these projects are 'Electronic signatures in message exchange services', 'Secure partner recognition', and 'Electronic financial transactions'.

We give the average execution times for selected routines on three different machines:

- The SIEMENS 7.541 computer is a 0.8 mips machine with an IBM-370 like instruction set. Key generation takes 40 seconds on the average, a signature 1.5 seconds and the authentication of a signature 0.3 seconds. All these numbers refer to a modulus of size 2^{512}. With such a modulus the hash function can process about 3500 characters of text per second.
- The EPSON PX-8 hand-held computer has a Z80 compatible microprocessor running on 2.45 Mhz. The RSA routines were written in Z80 assembler in connection with BASIC. We worked with a 256-bit modulus on this machine and got the following results: A signature takes 45 seconds and an authentication 3 seconds. (Note that the z80 processor has no hardware multiplication!)
- A third machine was the SIEMENS PC-X which is equipped with an INTEL 80186 microprocessor. Programming language was 'C' together with assembler subroutines. Using a modulus with 256 bits, signing takes 1.5 seconds and authenticating takes 0.2 seconds. The hash function can handle about 1700 characters per second. The programming for the EPSON was done by Michael Welschenbach, for the two SIEMENS machines by the author.

Acknowledgements

I would like to thank the staff at the GMD, especially B. Struif, E. Raubold and S. Herda, for providing excellent working conditions and helpful comments on an earlier draft of this paper, C.P. Schnorr for helping getting me started with the implementation and K. Keimel for constantly encouraging me in the course of writing this report.

References

[1] A.O.L. Atkin, R.G. Larson: On a Primality Test of Solovay and Strassen. *SIAM J. Comp.* 4 (1982), p. 789–791.

[2] D.W. Davies, W.L. Price: The Application of Digital Signatures Based on Public-Key Cryptosystems. *Proc. Int. Conf. on Comp. Comm.*, Atlanta 1980.

[3] D.W. Davies, W.L. Price: Digital Signatures - An Update. *Proc. Intl. Conf. on Comp. Comm.*, Sydney 1984, p. 845–849.

[4] J.M. DeLaurentis: A Further Weakness in the Common Modulus Protocol for the RSA Cryptoalgorithm. *Cryptologia* 8 (1984), p. 253–259.

[5] D.E. Denning: Digital Signatures with RSA and other Public-Key Cryptosystems. *Comm. ACM* 27 (1984), p. 388–392.

[6] W. Diffie, M.E. Hellman: New Directions in Cryptography. *IEEE Trans. Inform. Theory*, IT-22 (1976), p. 644–654.

[7] P. Erdös: On Almost Primes. *Am. Math. Monthly*, 57 (1950), p. 404–407.

[8] R.C. Fairfield, R.L. Mortenson, K.B. Coulthart: An LSI Random Number Generator. *Advances in Cryptology: Proc. of CRYPTO* 84, *Lect. Notes in Comp. Science* 196 (1985), p. 203–230.

[9] J. Gordon: Strong RSA Keys. *Electronic Letters*, 20 (1984), p. 514–516.

[10] G.H. Hardy, E.M. Wright: *An Introduction to the Theory of Numbers*. Oxford University Press, London 1954.

[11] D.G.N. Hunter: RSA Key Calculation in ADA. *The Comp. J.*, 28 (1985) p. 343–348.

[12] R.R. Jueneman, S.M. Matyas, C.H. Meyer: Message Authentication with Manipulation Detection Codes. *Proc. Symp. Security and Privacy*, 1983, p. 33–54.

[13] D.E. Knuth: *The Art of Computer Programming. Vol. 2: Seminumerical Algorithms*. 2nd. ed. Addison-Wesley 1981.

[14] D.J Lehman: On Primality Tests. *SIAM J. Comp.* 11 (1982).

[15] E.H. Michelman: The Design and Operation of Public-Key Cryptosystems. *Nat. Comp. Conf.* 1979, p. 305–311.

[16] H. Ong, C.P. Schnorr, A. Shamir: An Efficient Signature Scheme Based on Quadratic Equations. *Proc. of the 16th ACM Symp. on the Theory of Computing*, Washington 1984, p. 208–216.

[17] J.M. Pollard: A Monte Carlo Method for Factorization. *BIT* 15 (1975), p. 331–334.

[18] J.M. Pollard: Solution of $x^2 + ky^2 = m \pmod{n}$, with Application to Digital Signatures. Preprint 1984.

[19] C. Pomerance: Recent Developments in Primality Testing. *Math. Intelligencer*, 3 (1981), p. 97–105.

[20] C. Pomerance: On the Distribution of Pseudoprimes. *Math. Comp.*, 37 (1981), p. 587–593.

[21] C. Pomerance, J.L. Selfridge, S.S.Wagstaff: The Pseudoprimes to $25 \cdot 10^9$. *Math. Comp.* 35 (1980), p. 1003–1026.

[22] J.J. Quisquater, C. Couvreur: Fast Decipherment Algorithm for RSA Public-Key Cryptosystems. *Electronic Letters* 18 (1982), p. 905–907.

[23] M.O. Rabin: Probabilistic Algorithm for Testing Primality. *J. Number Theory* 12 (1980), p. 128–138.

[24] R.L. Rivest: Remarks on a Proposed Cryptoanalytic Attack on the MIT Public-Key Cryptosystem. *Cryptologia* 2 (1978), p. 62–65.

[25] R.L. Rivest: A Description of a Single-Chip Implementation of the RSA Cipher. *Lambda* 1980, p. 14–18.

[26] R.L. Rivest, A. Shamir, L. Adleman: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Comm. ACM* 21 (1978), p. 120–126.

[27] G.J. Simmons: A 'Weak' Protocol Using the RSA Cryptoalgorithm. *Cryptologia* 7 (1983), p. 180–182.

[28] G.J. Simmons, M.J. Norris: Preliminary Comments on the MIT Public-Key Cryptosystem. *Cryptologia* 1 (1977), p. 406–414.

[29] R. Solovay, V. Strassen: A Fast Monte-Carlo Test for Primality. *SIAM J. Comp.* 1 (1977), p. 84–85.

[30] H.C. Williams, B. Schmid: Some Remarks Concerning the MIT Public-Key Cryptosystem. *BIT* 19 (1979), p. 525–538.

[31] *OSIS – Open Shops for Information Services*. Final Report of the OSIS European Working Groups. Brussels 1985.