

A practical “perfect” pseudo-random number generator

Article submitted to Computers in Physics

on February 27, 1996

by Thierry Moreau,

president,
CONNOTECH Experts-conseils, Inc.

9130 Place de Montgolfier
Montreal, Qc
Canada, H2M 2A1

Tel.: 1-514-385-5691

Fax: 1-514-385-5900

e-mail: 74261.1240@compuserve.com

Abstract

The “ $x^2 \bmod N$ ” generator, also known as the BBS generator [2], has a strong theoretical foundation from the computational complexity theory and the number theory. Proofs were given that, under certain reasonable assumptions on which modern cryptography heavily relies, the BBS pseudo-random sequences would pass any feasible statistical test. Unfortunately, the algorithm was found to be too slow for computer simulation applications. In this article, we present a practical implementation of the “ $x^2 \bmod N$ ” generator. We show a variant of the Montgomery modular multiplication algorithm [21] tailored to the typical computer environment used for computer simulations. We observed an adequate level of performance for the “ $x^2 \bmod N$ ” generator to be seriously considered whenever an otherwise “good” pseudo-random generator casts a doubt about the results of a sensitive simulation.

1. Introduction

A pseudo-random generator (PR generator) is at the heart of any computer simulation program. Physics appears to be the application area that is most sensitive to the statistical quality of PR generators [28], [8]. In this paper, we present a practical implementation of the “ $x^2 \bmod N$ ” generator, also known as the BBS generator, from the names of authors of [2]. The statistical quality of the “ $x^2 \bmod N$ ” generator is supported by decisive theoretical results of the complexity theory and the number theory. But up to now, the implementation was not considered practical, mainly for performance considerations [17], [25] page 418. So our findings represents an alternative solution to the statistical uncertainty that may be encountered in a simulation application: at the expense of a bearable performance penalty, use a PR generator with a proven statistical quality.

In the next two section, we describe the “ $x^2 \bmod N$ ” generator. Section 4 describes the Montgomery multiplication, the optimization technique that we use to implement a practical version of the “ $x^2 \bmod N$ ” generator. A slight variation from the original “ $x^2 \bmod N$ ” specification is explained and justified in section 5. In a following section, the implementation details are related. In the appendix, we include the pseudo-code for the run-time portion of the proposed pseudo-random generator.

2. Theoretical foundations of the “ $x^2 \bmod N$ ” generator

The theory supporting the “ $x^2 \bmod N$ ” generator was developed during the 1980-1986 period, when rapid advances were made in the new field of public key cryptography. In a sense, Andrew C. Yao, in [31], expanded the Shannon's theory of information [27] to the trap-door one-way functions used in cryptography. Yao defined a polynomial statistical test in terms of computational complexity theory, and defined a perfect generator as one which passes every such tests. According to these criteria, the best generators to date are not yet perfect: their proofs are based upon unproven assumptions about the difficulty of solving two hard problems in the number theory: the discrete logarithm problem [26], and the factorization problem (see the fine print note at the end of [2]). It should be noted, though, that either one of these problems is the cornerstone of every public key cryptosystem in use today. As a consequence, much effort has been devoted to find “efficient” algorithms to either solve these problems, or support empirical evidence about their difficulty with formal proofs. Neither of these research directions was fruitful. The “ $x^2 \bmod N$ ” generator statistical quality is based on the difficulty of factoring the large parameter N .

The significance of this theoretical foundation is two-fold. On one hand, the complexity theory criterion seems way above the practical requirements of computer simulation applications. Assume a moment that a major breakthrough was to occur in factorization algorithms. This would void the proof that the “ $x^2 \bmod N$ ” generator passes every polynomial-time statistical test, but would not provide such a test. Even if such a test was found, expectably it would be very specific, and thus harmless in practice for computer simulation applications.

On the other hand, the theorems in the computational complexity theory reads like “for any integer t and for all but a fraction δ of parameters N , there is always a length $n=|N|$ large enough for which the ‘ $x^2 \bmod N$ ’ generator fails any statistical test with a probability less than $1/n^t$.” This raises two questions: i) how large n should be, and ii) how can we avoid special parameters N which fall into the “fraction δ of parameters N ” for which the theory is inapplicable? Having set an extreme criterion for perfectness, the theory less useful when it comes to implementation details. Based on rationales for recommended practice in the area of cryptography [1] and review of current practice in PR generators for simulation, we advocate a parameter N commensurate with the period of “good” PR generators used in simulation, that is $2^{150} \leq N \leq 2^{300}$ (we will see that the period of the “ $x^2 \bmod N$ ” generator is $\approx N/8$ under certain circumstances). In other words, we feel confident enough in the theoretical foundation of the “ $x^2 \bmod N$ ” generator to let other design requirements dictate the minimum size of N . To address the issue of hypothetical unsatisfactory parameters N , we offer a choice among 1 million different values. A simulation application may be run with a different N each time.

3. The “ $x^2 \bmod N$ ” generator

Here is the specification for the “ $x^2 \bmod N$ ” generator. Let N be a Blum integer, that is $N=P \times Q$, where P and Q are large prime numbers, $P \neq Q$, $P \bmod 4=3$, $Q \bmod 4=3$. Using the terminology of definition 1 from [14], a state of the generator is represented by the triplet $\langle N, k, x_i \rangle$; the initial state $\langle N, k, x_0 \rangle$ is defined by a proper selection of parameters N and k , and by setting $x_0 = x^2 \bmod N$ for some integer x less than N and relatively prime with it; the transition function is defined by $x_{i+1} = x_i^2 \bmod N$; the output function is defined by $u_i = x_i \bmod 2^k$. The “ $x^2 \bmod N$ ” generator is purely periodic: for any state $\langle N, k, x_i \rangle$, the generator returns to this exact same state after the transition function is applied a number of times, this number being the period π (so $x_{i+\pi} = x_i$).

A comprehensive account of this generator is found in [2]. A good introductory text is in [4]. Originally, k was set to 1. Vazirani and Vazirani, in [29], showed that it could be increased up to

$k \leq \log_2(\log_2(N)) = \log_2(|N|)$ without compromising the statistical quality of the generator. Furthermore, reference [29] concludes by a relation between k and empirical findings about the difficulty of the factorization problem: say a solution requires $O(f(N))$ operations, then $k \leq \log_2(f(N))$.

To decide on a value for k , we first reviewed references [29] and [26], and accounts of current factorization algorithms [9]. It became evident that a reasonable upper limit for the parameter k is an issue for which the theoretical answers are limited. Then we plotted the factorization complexity on a logarithmic scale, with the corresponding values of k , for the various sizes of N . The value of $k=24$ for $|N|=180$ is suggested as a reasonable choice, which lies clearly below the curve representing the best factorization. Further empirical studies are needed to improve the understanding of the parameter k . One can speculate that $k=|N|/2$ would still provide a generator with good statistical behavior.

Williams was the first to apply the useful properties of Blum integers to cryptography [30]. See also [18] for early work done independently of [2]. The Williams proposal is now embedded in an international standard for digital signatures [12], [10]. The “ $x^2 \bmod N$ ” generator has been proposed for an efficient probabilistic public key encryption scheme, see [3], and for automated session key establishment with implicit authentication, see [22]. Other cryptographic applications include [5] and [19].

Some properties of the “ $x^2 \bmod N$ ” generators are more specially relevant to simulation applications. Let's review them, without any proof given in [2].

- Let N be a Blum integer with the additional property that $P_1=(P-1)/2$, $P_2=(P-3)/4$, $Q_1=(Q-1)/2$, and $Q_2=(Q-3)/4$ are all prime numbers, and that “2 is a quadratic residue with respect to at most one of P_1 and Q_1 ” (in other words, x_p and x_q do not exist for $2=x_p^2 \bmod P_1$ and $2=x_q^2 \bmod Q_1$). Then, the period of the “ $x^2 \bmod N$ ” generator is a divisor of $2P_2Q_2$, and it exists some x_0 for which this period is exactly $2P_2Q_2$ (theorem 8 of [2]). As a consequence, the period is one of $1, 2, P_2, 2P_2, Q_2, 2Q_2, P_2Q_2$, or $2P_2Q_2$.
- For any Blum integer N (and a fixed k), there are exactly $4P_2Q_2+2P_2+2Q_2+1$ possible states for the generator. With a large N satisfying the conditions of the above paragraph, nearly 1/2 of them are in a cycle with a maximum period equal to $2P_2Q_2$ (actually, we noticed a probability close to 1 for a seed to produce the longest period for such N 's).
- For any Blum integer N , there is a relatively efficient algorithm to jump ahead in the sequence of states: $x_{i+t} = x_i^e \bmod N$, where $e = 2^t \bmod 2P_1Q_1$ (theorem 10a of [2]). This gives the “ $x^2 \bmod N$ ” generator a direct access capability in the output sequence. We will show why it is preferable for the modulus to be odd. For now, we notice $e = 2^t \bmod 2P_1Q_1 = 2 \times (2^{t-1} \bmod P_1Q_1)$, which gives an odd modulus.

4. An optimization technique used in cryptography

Extreme computing requirements has been a significant impediment to the widespread use of public key cryptography. Among the many optimization proposed to decrease the processing requirements, the Montgomery multiplication [21] has reportedly been applied to special-purpose hardware [7] and to digital signal processors [6], none of which is readily adapted to the traditional scientific computer environment. In this section, we present the Montgomery multiplication from a functional perspective, and then we present the algorithm in pseudo-code.

Montgomery multiplication allows fast modular arithmetic for a modulus N relatively prime to B , where $B > N$, and arithmetic modulo B is easy. Since we are interested in the multiple precision case, we let $B = b^n$ (e.g. $2^{127} < N < 2^{128}$, N is odd, $b = 2^{32}$, $n = 4$, so $B = 2^{128}$). Given T satisfying $0 \leq T < BN$, the Montgomery reduction algorithm efficiently computes $T B^{-1} \bmod N$. Our focus is the modular multiplication, so let $T = x'y'$ where $x' = xB \bmod N$ and $y' = yB \bmod N$; then the Montgomery reduction algorithm efficiently computes

$x'y'B^{-1} \bmod N = xyB \bmod N$.

Let us use the notation $M_{N,B}(X,Y)$ for the result of $XYB^{-1} \bmod N$, using the Montgomery reduction algorithm. Then $xyB \bmod N = M_{N,B}(xB \bmod N, yB \bmod N)$. To convert an integer x , $0 \leq x < N$, into $xB \bmod N$, compute $M_{N,B}(x, B^2 \bmod N)$. The value $B^2 \bmod N$ should be pre-computed once and for all; this must be done with a general purpose division operation. To recover x from $xB \bmod N$, compute $M_{N,B}(xB \bmod N, 1)$. There are two routes to complete a single modular multiplication: $M_{N,B}(M_{N,B}(x,y), B^2 \bmod N) = xy \bmod N$, or by pre-computing $x' = M_{N,B}(x, B^2 \bmod N)$ and $y' = M_{N,B}(y, B^2 \bmod N)$, and then $M_{N,B}(M_{N,B}(x', y'), 1) = xy \bmod N$. Whenever a series of multiplications is performed on a same set of inputs or intermediate results, the latter route is more efficient. This is the case of the “ $x^2 \bmod N$ ” generator.

We refer the reader to references [21], [6], and [7] for formal justifications and proofs for the algorithm we are about to describe. Our multi-precision algorithm is a variant of the one presented in [6]. The algorithm for $M_{N,B}(X,Y)$ is introduced by figure 1 using a mathematical notation, with no consideration of multi-precision arithmetic. We first need to compute N' such that $BB^{-1} - NN' = 1$, where $BB^{-1} \bmod N = 1$. In the general case, this can be done using the generalized Euclid algorithm and then adjusting the signs of the returned values. The trivial division R/B is always an exact division. The trivial modulo reduction is implemented by a comparison and a subtraction. This is illustrated by a numerical example (figure 2) which introduces the multi-precision case.

In the multiprecision algorithm (figure 3), capital letters are multi-precision variables. The indices are as expected for natural integers, e.g. $X = \sum_i X_i \times b^i$. The algorithm is an application of the convolution-sum method for the multiplication. This is pictorially represented in the example by the diagonals in the lower-left to upper-right direction. These diagonals are added sequentially (e.g. $\sum_i X_i \times Y_{k-i}$), starting with the least significant portion of the product. References [21] and [7] do not use the convolution-sum. Instead, they process digits column-wise (e.g. $T = T + \sum_i X_k \times Y_i \times b^{k+i}$). In the algorithm of figure 3, the variable c is an accumulator with sufficient capacity for a sum of products and multiple carry bits from the additions (up to $2n$ of them). This is a common arrangement for digital signal processors and dedicated integrated circuits (e.g. a multiply-accumulate instruction that accumulates 32 bit products in a 40 bit accumulator). Our implementation is limited by the simpler multiply instructions of general purpose CPUs. So we decided on $b=2^{30}$, with accumulation of 60 bit products in 64 bits.

There is a significant optimization in the multi-precision algorithm. It avoids most of the computation of $T \times N' \bmod B$ by exploiting the following observation: the R_0 value depends on T and Q_0 , but since $R_0 = 0$, we can set Q_0 as the unknown. Likewise, Q_{i+1} recursively depends on T and Q_i for $0 < i \leq k$. For the number-theoretic justification, see the references. As an interesting consequence of this optimization, only the least significant part of N' is needed (notation N'_0). We reproduce, in figure 4, the simple algorithm of [6] to efficiently find N'_0 from N_0 and b , when b is an exact power of two. Obviously, N'_0 can be computed once and for all.

Our algorithm computes R/B directly, without ever storing the full representation of R . As a final remark, R_n is actually a local storage area of the algorithm (like Q and lower case variables). Consequently, the storage requirement for R is the same as for X and Y .

5. A variant of the “ $x^2 \bmod N$ ” generator

From the above account of the Montgomery multiplication, it should be clear how the “ $x^2 \bmod N$ ” generator can be made more efficient: change the output function from $u_i = x_i \bmod 2^k$ to $u_i = x_i B \bmod 2^k$. Formally, the transition function remains $x_{i+1} = x_i^2 \bmod N$, but the state x_i is now stored as $x_i B \bmod N$. Then,

the transition function is efficiently implemented from the equality $M_{N,b,n}(x_i \mathbf{B} \bmod N, x_i \mathbf{B} \bmod N) = x_{i+1} \mathbf{B} \bmod N$. Here, \mathbf{B} is an exact power of 2 greater than \mathbf{N} and chosen according to section 4; it is part of the PR generator specification. The values \mathbf{b} and \mathbf{n} are such that $\mathbf{B} = \mathbf{b}^n$; their choice has no effect on the output of the generator (hence the generator is portable across computer architectures with different word sizes).

How does the change in the output function affect the statistical quality of the generator? Nothing in the theory points towards any impact. In the proof of lemma 3 in [2], the distribution of $x^2 \mathbf{B} \bmod N$ is said to be uniform. So it would be a waste of computing resources to specify a conversion from $x_i \mathbf{B} \bmod N$ to $x_i \bmod N$ before taking the \mathbf{k} least significant bits in the output function.

6. Implementation strategy

The “ $x^2 \bmod N$ ” generator is an algorithm commanded by the theory (with other good PR generators, there are more interactions between software engineering and theoretical developments). From a software engineering perspective, the core algorithm requires the serious optimizations that are made possible with the Montgomery multiplication algorithm. Also, the parameters must be set with due respect to the theory. Ascertaining the period length is another aspect of the algorithm driven by the theory.

We found that a practical implementation of the “ $x^2 \bmod N$ ” generator naturally breaks down in three parts: i) the utmost optimization of the Montgomery multiplication applied to the generator, ii) prime number generation, and iii) a run-time set of procedures.

6.1 Optimized Montgomery multiplication

The algorithm of figure 3 consists of indice computations and “payload” computations (the multiplications, additions, comparisons, subtractions, and divisions by exact powers of two). Luckily, all indice computations are driven by an implementation constant, \mathbf{n} . This is a favourable circumstance for preprocessing, where a program generates a program: a first version of the algorithm generates individual payload instructions in program source code instead of executing them (like the compiler optimization technique known as loop expansion). The payload computations are centered on the accumulator variable \mathbf{c} which handles extended precision and carries. This is specific to CPU architectures and directly accessible in assembly language. Accordingly, the preprocessing step generates assembler language source code. Actually, there is an optional intermediate step where high level source code is generated, for verification and for checking the conventions for parameter passing and memory addressing. For $\mathbf{n} = 6$, the generated code has 112 lines of elementary payload instructions with the high level version. This turned into 362 generated assembler instructions (with the Intel 386 instruction set). This encompasses a further optimisation to the algorithm of figure 3: from the equality $\mathbf{X} = \mathbf{Y}$, it is possible to replace two multiplications by one multiplication and a shift to the left by one bit position. In this context, it is valid and convenient to store the result in the same variable as the argument (that is \mathbf{X} , \mathbf{Y} , and \mathbf{R} are all the same variable).

With this scheme, it was possible to fully exploit the 32 bits architecture by multiplying 30 bit numbers and accumulating them in a 64 bits accumulator \mathbf{c} , leaving 4 bits for carries. Consequently, $\mathbf{b} = 2^{30}$. We decided on $\mathbf{n} = 6$ to get $\mathbf{B} = 2^{180}$, and allow \mathbf{N} in the range $2^{179} < \mathbf{N} < 2^{180}$. The period of the PR generator is dependent on number-theoretic characteristics of the parameter \mathbf{N} , and on the particular seed \mathbf{x}_0 .

6.2 Prime number generation

In all cases, the parameter \mathbf{N} should be a Blum integer. It means that $\mathbf{N} = \mathbf{P} \times \mathbf{Q}$ with \mathbf{P} and \mathbf{Q} both prime numbers congruent to 3 modulo 4. To facilitate the exact measurement of the period of the generator, a

special form for \mathbf{N} is preferred (the form prescribed by theorem 8 in [2], as explained in section 3 above). Furthermore, the theoretical foundation of the “ $x^2 \bmod N$ ” generator leaves open the possible existence of some special \mathbf{N} for which the generator would not be as good as predicted by the theory (even though such \mathbf{N} might pass every practical statistical test). To let the user of the proposed generator convince himself that a particular simulation result is not hit by an unexpectedly bad \mathbf{N} , we recommend a choice of \mathbf{N} among more than one million possibilities.

The selection of a candidate for \mathbf{P} or \mathbf{Q} is the usual problem of large prime number selection. For cryptography, there are two approaches to this task: repeatedly use a compositeness test for a candidate integer until you believe that the number is not composite, or build a proven prime [20]. The latter approach is a new development not easily applicable for our purpose (due to the restriction on \mathbf{N} to ascertain the period). A good compositeness test uses a random integer about the same size as the integer under scrutiny and reveals its compositeness with a probability of 1/2 or 3/4 (assuming it is composite). The Miller-Rabin compositeness test is the most widely used in cryptography [24], [23]. If you use the Miller-Rabin test t times on a composite number, you have a less than 4^{-t} probability of falsely thinking it is a prime.

Using tools developed for cryptography, we found numbers \mathbf{P} such that \mathbf{P} , $\mathbf{P}_1=(\mathbf{P}-1)/2$, and $\mathbf{P}_2=(\mathbf{P}-3)/4$ are all (probabilistically) prime. We ran the Miller-Rabin test in parallel for \mathbf{P} , \mathbf{P}_1 , and \mathbf{P}_2 , aborting the search as soon as one of them was found to be composite (incidentally, for this purpose, random number generation was driven by a generator proposed by L'Écuyer [15]). We also rejected any candidate for which 2 was a quadratic residue with respect to $\mathbf{P}_1=(\mathbf{P}-1)/2$ (let $x=2^e \bmod \mathbf{P}_1$, where $e=(\mathbf{P}_1+1)/4$, then 2 is a quadratic residue if and only if $x^2 \bmod \mathbf{P}_1=2$). In view of the inherent complexity, we recommend not to include prime number generation in a production version of our generator. We created a table of 1449 (probabilistically) prime numbers \mathbf{P}_2 , each in the range $3 \times 2^{174} < \mathbf{P}_2 < 4 \times 2^{174}$. This gives over one million possibilities for composite numbers $\mathbf{N}=(4\mathbf{P}_2+3)(4\mathbf{Q}_2+3)$, indeed exactly $1449 \times 1448/2 = 1\,049\,076 = 2^{20} + 500$.

6.3 Run-time procedures

One of the goal for the implementation strategy is to minimize the number of different operations on large numbers that are used in the run-time procedures (to reduce software development, verification, and maintenance overhead). The single operation where optimization is critical is the very PR generator transition function. The “ $x^2 \bmod N$ ” generator has a relatively efficient algorithm for jumping ahead. This direct access capability is required for to ascertain the period length. Fortunately, the Montgomery multiplication is an optimization technique well suited to the direct access capability of the “ $x^2 \bmod N$ ” generator. So we optimize this aspect of the PR generator as well, but to a lesser extent than for the transition function. The fast exponentiation function used for this purpose is a classical algorithm [13].

The pseudo-code of the runtime procedures is given in the appendix. It starts with the description of the data types. A complete list of operations on multi-precision integers is given. It shows that the run-time procedures do not require a full multi-precision package (e.g. addition of two multi-precision integers is never used). But there is a requirement for general purpose multi-precision divisions, and we refer the reader to [11] for a complete algorithm. It is wise to program this long division in a high level language. The algorithm in [11] requires a different base than the Montgomery multiplication (we used $\mathbf{b}=2^{10}$). There are actually four representations for multi-precision integers, the two other being the external (ASCII) representation and the double multi-precision (values up to \mathbf{B}^2) used for temporary products without overflow. Care was taken to make data type conversions explicit in the pseudo-code.

6.4 User's view of the implementation

To set-up the initial state of the generator, just call the procedure “setup_parameters” with an argument

between 0 and 1000000 (or 2^{20}), and then the procedure "setup_seed" according to the pseudo-code explanations. Thereafter, each call to the procedure "BBS_step" will return a pseudo-random 24 bit integer from the uniform probability distribution. The reader is referred to section 8 of reference [14] for applications where multiple generators must be run in parallel. Our generator is well suited to these applications, either by using different parameters for each generator or by using an array of global variables X_i and either setting up different random seeds (with the procedure "setup_seed") or deterministically using the direct access capability of the generator (with the procedure "jump_forward").

7. Empirical observations

[3× performance degradation]

[availability: within a crypto project, C++ code, limited]

8. Conclusion

We presented a pseudo-random number generator "fit for all applications" [31], a "potential breakthrough" [16]. We reported the many steps that were taken to make the theory work with a reasonable performance, in the typical computer environment where simulation applications are run. Notably, we showed the use of the Montgomery multiplication, a technique that could be useful in other PR generators using modular arithmetic, allowing the designer to break the barrier of 32 bits arithmetic with a reduced performance penalty.

Advances in science often come from the application of developments in one field to solve a difficulty in another field. The " $x^2 \bmod N$ " generator is number-theoretic unpredictable. The practical implementation reported here can be used to raise confidence in simulation results beyond what was previously possible with good generators and extensive statistical testing of them.

References

- [1] Beth, Th., Frish, M., Simmons, G.J. (editors), Public Key Cryptography: State of the Art and Future Directions, Lecture Notes in Computer Science, no. 578, Springer-Verlag, 1991
- [2] Blum, L., Blum, Manuel, and Shub, M., A Simple Unpredictable Pseudo-random Number Generator, SIAM Journal of Computing, vol. 15, no. 2, May 1986, pp 364-383
- [3] Blum, Manuel, and Goldwasser, Shafi, An Efficient Probabilistic Public-key Encryption Scheme which Hides All Partial Information, In Advances in Cryptology: Proceedings of Crypto'84, Springer-Verlag, 1985, pp 289-299
- [4] Brassard, Gilles, Modern Cryptology, a Tutorial, Lecture Notes in Computer Science, no. 325, Springer-Verlag, 1988
- [5] Brassard, Gilles, On Computationally Secure Authentication Tags Requiring Short Shared Secret Keys, in Advances in Cryptology, Proceedings of Crypto'82, Plenum Press, New York, 1983
- [6] Dussé, Stephen R., and Kaliski, Burton S. Jr., A Cryptographic Library for the Motorola DSP56000, Advances in Cryptology, Eurocrypt'90, Lecture Notes In Computer Science no. 473, pp 230-244, Springer-Verlag, 1990

- [7] Eldridge, S. E., and Walter, Colin D., Hardware Implementation of Montgomery's Modular Multiplication Algorithm, IEEE Transactions on Computers, Vol. 46, no. 6, June 1993, pp 693-699
- [8] Ferrenberg, Alan M., Landau, D.P., and Wong, Y. Joanna, Monte Carlo Simulations: Hidden Errors from "Good" Random Number Generators, Physical Review Letters, Vol. 69, Number 23, 7 December 1992, pp. 3382-3384
- [9] Fahn, Paul, RSA Laboratories, Answers to Frequently Asked Questions About Today's Cryptography, revision 2.0, RSA Data Security, Inc., October 1993
- [10] Guillou, Louis Claude, Quisquater, Jean-Jacques, Walker, Mike, Landrock, Peter, and Shaer, Caroline, Precautions taken against various potential attacks in ISO/IEC DIS 9796 'Digital signature scheme giving message recovery', Advances in Cryptology, Eurocrypt'90, Lecture Notes In Computer Science no. 473, pp 465-473
- [11] Hansen, Per Brinch, Multiple-length Division Revisited: a Tour of the Minefield, Software Practice and Experience, vol. 24(6), June 1994, pp 579-601
- [12] ISO/IEC 9796:1991, Information Technology - Security Techniques - Digital Signature Scheme Giving Message Recovery
- [13] Knuth, D.E., The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Addison-Wesley, 1969
- [14] L'Écuyer, Pierre, Uniform Random Number Generation, Annals of Operations Research, vol. 53 (1994), pp 77-120
- [15] L'Écuyer, Pierre, Combined Multiple Recursive Random Number Generators, Les cahiers du GERAD, G-95-15, March 1995
- [16] L'Écuyer, Pierre, personal conversation
- [17] L'Écuyer, Pierre, and Proulx, René, About Polynomial-Time "Upredictable" Generators, Proceedings of the 1989 Winter Simulation Conference. pp 467-476
- [18] Lieberherr, Karl, Uniform Complexity and Digital Signatures, Theoretical Computer Science, Vol. 16 (1981), pp 99-110
- [19] Lim, Chae Hoon, and Lee, Pil Joong, Another Method for Attaining Security Against Adaptively Chosen Ciphertext Attack, Advances in Cryptology, Crypto'93, LNCS no. 733, Springer Verlag 1993, pp 420-434
- [20] Maurer, Ueli M., Fast Generation of Secure RSA-Moduli with Almost Maximal Diversity, in Advances in Cryptology, Eurocrypt '92, Lecture Notes in Computer Science, no. 658, Springer-Verlag, 1992, pp 636-647
- [21] Montgomery, Peter L., Modular Multiplication Without Trial Division, Mathematics of computations, Vol. 44, no. 170, April 1985, pp 519-522
- [22] Moreau, Thierry, Probabilistic Encryption Key Exchange, Electronics Letters, Vol. 31, number 25,

7th December 1995, pp 2166-2168

- [23] National Institute of Standards and Technology (NIST), Digital Signature Standard, Federal Information Processing Standards Publication FIPS PUB 186, U.S. Department of Commerce, May 1994
- [24] Rabin, M.O., Probabilistic algorithm for testing primality, *Journal of Number Theory*, v 12 (1980), pp. 128-138
- [25] Schneier, Bruce, *Applied Cryptography*, 2nd edition, John Wiley & Sons, 1996
- [26] Schriff, A.W., and Shamir, Adi, The Discrete Log is Very Discrete, *Proceedings of the 22nd ACM symposium on theory of computing*, ACM press, 1990, pp 405-415
- [27] Shannon, C. E., *Communication Theory of Secrecy Systems*, *Bell System Technical Journal*, 28 (1949) pp 656-715
- [28] Vattulainen, I., Ala-Nissila, T., and Kankaala, K., Physical Tests for Random Numbers in Simulation, *Physical Review Letters*, Vol. 73, Number 19, 7 November 1994, pp. 2513-2516
- [29] Vazirani, Umesh V., and Vazirani, Vijay V., Efficient and Secure Pseudo-random Number Generation, *Proceedings of the 25th IEEE Symposium on the Foundations of Computer Science*, 1984, pp 458-463
- [30] Williams, Hugh C., A Modification of RSA Public-Key Encryption, *IEEE Transactions on Information Theory*, Vol IT-26, no. 6, November 1980, pp 726-729
- [31] Yao, Andrew C., Theory and Application of Trapdoor Functions, *IEEE 23rd Symposium on Foundations of Computer Science*, 1982, pp 80-91

Explanation	Operation
<p>To compute $M_{N,B}(X,Y)$, you do</p> <p style="padding-left: 100px;">a full multiplication,</p> <p style="padding-left: 20px;">half of a multiplication (B is an exact power of 2),</p> <p style="padding-left: 100px;">a full multiplication plus an addition,</p> <p style="padding-left: 20px;">a trivial division (B is an exact power of 2),</p> <p style="padding-left: 20px;">and a trivial modulo reduction ($R/B < 2N$),</p> <p style="padding-left: 100px;">and then you return:</p>	<p>$T = X \times Y,$</p> <p>$Q = T \times N' \bmod B,$</p> <p>$R = Q \times N + T,$</p> <p>$R/B,$</p> <p>$R/B \bmod N,$</p> <p>$R/B \bmod N = XYB^{-1} \bmod N.$</p>

Figure 1) The Montgomery multiplication, mathematical notation

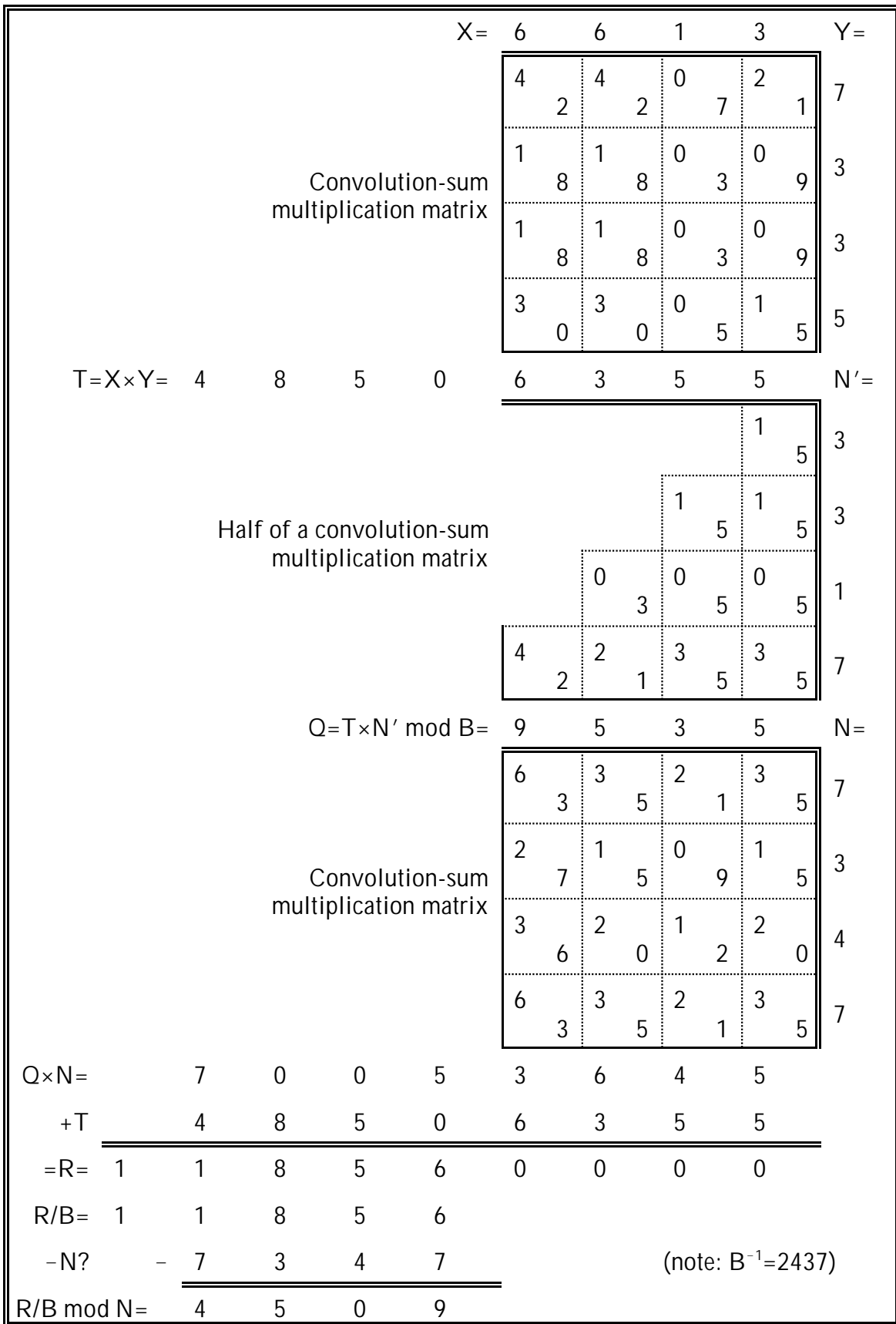


Figure 2) The Montgomery multiplication, an example: $M_{7347,10000}(6613,7335)=4509$

```

MN,b,n(X, Y)
  N0' := b - (N0-1 mod b);
  c := 0;
  for k ← 0 to n-1 do
    c := c + ∑0 ≤ i < k (Xi × Yk-i + Qi × Nk-i);
    c := c + Xk × Y0;
    Qk := c × N0' mod b;
    c := c + Qk × N0;
    c := c/b;
  for k ← 0 to n-1 do
    c := c + ∑k < i < n (Xi × Yn+k-i + Qi × Nn+k-i);
    Rk := c mod b;
    c := ⌊c/b⌋;
  Rn := c;
  if R ≥ N then
    R := R - N;
  return R;

```

Figure 3) The Montgomery multiplication, multi-precision algorithm

```

modular_inverse(N, b)
  Let k be such that b = 2k.
  t := 1;
  for i ← 2 to k do
    if (N0 × t mod 2i) ≥ 2i-1
      t := t + 2i-1;
  return b - t;

```

Figure 4) A method to find $N_0' = b - (N_0^{-1} \bmod b)$ given N_0 and b , an exact power of 2

Appendix: Pseudo-code for the run-time portion of the proposed generator

Preliminaries

Data types for integer representations

- integer integer variable, native representation of the target computer
- small_int small integer $0 \leq a < b$
- multi multi-precision integer $0 \leq a < B$ for direct manipulation
- e_multi extended multi-precision integer $0 \leq a < B^2$ for products and dividends
- multi_M multi-precision integer $0 \leq a < B$ for Montgomery multiplication
note: multi_M is independent of whether a or $aB \bmod N$ is represented
- multi_asc ASCII representation of multi-precision integer $0 \leq a < B$

Explicit conversions (used as function names in the pseudo-code):

- small_int_To_multi
- small_int_To_multi_M
- multi_asc_To_multi
- multi_To_multi_M
- multi_M_To_multi
- multi_To_e_multi
- multi_To_e_multi

List of elementary operations on multi-precision values:

- Multi-precision multiplication:
multi \times multi \rightarrow e_multi
note: when the product is used as a multi, truncation to product $< B$ is implicit
- Integer multi-precision division with remainder
[e_multi/multi] \rightarrow multi
e_multi mod multi \rightarrow multi
note: quotient always $< B$
- Addition
multi + small_int \rightarrow multi
note: sum $< B$
- Substraction
multi - small_int \rightarrow multi
note: difference ≥ 0
- Comparaison of two integers $< B$ for equality.
multi_M \neq multi_M
multi \neq 0
- Testing if a bit of weighth 2^i is set in the binary representation of an integer $< B$
(multi.AND. 2^{integer}) \neq 0
- Generating the representation of B
 \rightarrow e_multi

List of numerical functions

- Dussé-Kalinski inverse computation (figure 4)
multi_M \rightarrow small_int
- Generic Montgomery multiplication, local function $M_{N,b,n}(X,Y)$
multi_M \times multi_M \rightarrow multi_M

- note: this function refers to a structure describing the modulus characteristics
- Montgomery squaring for the “ $x^2 \bmod N$ ” generator, function BBS_step()
current state→integer
 - Fast modular exponentiation algorithm, local function fast_mod_exp(A,E,N)
multi_M^{multi}→multi_M
note: this function refers to a structure describing the modulus characteristics

Pseudo-code

global constants

```
log2_b=30,
b=2log2_b,
n=6,
e_multi B=bn,
k=24;
```

global variables

```
multi      P, P2,
multi      Q, Q2,
multi      Nmulti,
```

```
multi_M     Xi;
```

structure

```
multi_M     m;
small_int   inverse,
multi_M     B2_mod;
end_structure      N, P1Q1;
```

local function modular_inverse(N)

```
input      multi_M N;
```

```
output     small_int required for the Montgomery multiplication algorithm;
```

description

Use the Dussé-Kalinski algorithm of figure 4, with the constant b specified above.

end

local procedure setup_P2_Q2(i)

```
input      integer i,
```

```
constant   s=1449;
```

```
multi_asc Primes[s]; { A table of 1449 pre-computed primes }
```

begin

```
local variables integer ix, iy;
```

```
{ from 0≤i<s(s-1)/2, find <ix,iy> where 0≤ix<iy<s }
```

```
ix:=i mod (s-1)/2;
```

```
iy:=⌊i/((s-1)/2)⌋;
```

```
if (iy<((s-1)/2)).AND.(ix≥iy) then
```

```
begin
```

```
ix:=(s-2)-ix;
```

```
iy:=(s-1)-iy;
```

```

    end
    P2:=multi_asc_To_multi(Primes[ix]);
    Q2:=multi_asc_To_multi(Primes[iy]);    { P2<Q2<2P2<2Q2 }
end

```

```

public procedure setup_parameters(i)
input  integer i;    { a (possibly random) indice in the set of 1449×1448/2=1 049 076=220+500
                    candidate N, i≥0 }
begin
    local variables    multi P1, Q1,
                      multi x;
    call setup_P2_Q2(i);

    P1:=small_int_To_multi(2)×P2+1;
    P:=small_int_To_multi(2)×P1+1;
    Q1:=small_int_To_multi(2)×Q2+1;
    Q:=small_int_To_multi(2)×Q1+1;
    Nmulti:=P×Q;

    { prepare MN,b,n() ,including pre-computation of B2 mod N }
    N.m:=multi_To_multi_M(Nmulti);
    N.inverse:=modular_inverse(N.m);
    x:=B mod Nmulti;
    N.B2_mod:=multi_To_multi_M((x×x) mod Nmulti);

    { prepare MP1Q1,b,n() , including pre-computation of B2 mod (P1×Q1) }
    P1Q1.m:=multi_To_multi_M(P1×Q1);
    P1Q1.inverse:=modular_inverse(P1Q1.m);
    x:=B mod (P1×Q1);
    P1Q1.B2_mod:=multi_To_multi_M((x×x) mod (P1×Q1));

end

```

```

local function MN,b,n(X,Y)
inputs multi_M X,
        multi_M Y,
output  multi_M R;
description
    Apply algorithm from figure 3 with the following adaptations:
    - the algorithm N is the variable N.m, and
    - N0' is precomputed either in the variable N.inverse.
end

```

```

local function fast_mod_exp(A,E,N)
inputs multi_M A,    { a base }
        multi E,     { an exponent }
        structure N; { a modulus descriptor }
output  multi_M representation of the expression (AE)B mod N.m;
begin
    local variables    multi_M R,

```

```

integer i;

R:=MN,b,n(small_int_To_multi_M(1),N.B2_mod); { simply 1 in the proper representation }
for i←(n×log2_b-1) downto 0 do
begin
  R:=MN,b,n(R,R);
  if (E.AND.2i)≠0 then
  begin
    R:=MN,b,n(R,A);
  end
end
return R;
end

```

```

public function BBS_step()
output a random integer between 0 and 2k-1 inclusive
description
  Apply algorithm from figure 3 with the following adaptations:
  - the algorithm N is the variable N.m,
  - N0' is precomputed either in the variable N.inverse,
  - X, Y, and R are all three associated with multi_M variable Xi, and
  - the returned value is no longer R; it is R0 mod 2k.
end

```

```

public function jump_forward(T) { may be used directly to set the Xi variable, as in
                               Xi := jump_forward(multi_asc_To_multi("1000000000000")); }
input      multi T; { the number of states to jump forward from current one }
output     the multi_M representation of (Xie)B mod N.m for e=2T;
assumption T ≥ 1
begin
  local variables      multi_M x;
                      multi E;
  x:=MP1Q1,b,n(small_int_To_multi_M(2),P1Q1.B2_mod); { simply 2 in the proper representation }
  x:=fast_mod_exp(x,T-1,P1Q1);
  E:=multi_M_To_multi(MP1Q1,b,n(x,small_int_To_multi_M(1)));
  return fast_mod_exp(Xi,2×E,N);
end

```

```

public procedure setup_seed(S)
input multi S;      { a (possibly random) seed indice, S<N }
begin
  local variable multi X;
  X := S;
retry: if (X mod P)=0 then X := (X+1) mod N; { not a multiple of P }
      if (X mod Q)=0 then
      begin
        X := (X+1) mod N; { hit a multiple of Q, try something else }
        goto retry;
      end
end

```

```

Xi := MN,b,n(multi_To_multi_M(X), N.B2_mod);
call bbs_step;
if (Xi = MN,b,n(small_int_To_multi_M(1), N.B2_mod))
.OR. (Xi = jump_forward(small_int_To_multi(2)))
.OR. (Xi = jump_forward(P2) { P2 < Q2 < 2P2 < 2Q2 }
.OR. (Xi = jump_forward(Q2))
.OR. (Xi = jump_forward(2×P2))
.OR. (Xi = jump_forward(2×Q2))
.OR. (Xi = jump_forward(P2×Q2)) then
begin
  X := (X+1) mod N; { hit a period shorter than the maximum, try something else }
  goto retry;
end
if Xi ≠ jump_forward(2×P2×Q2)) then
begin
  { Serious difficulty, either with N or with the implementation }
  abort;
end
end
end

```
