Cryptography, Statistics and Pseudo-Randomness (Part II)

Stefan Brands* Richard Gill†

Abstract

This paper is a sequel to Brands and Gill (1995), which contained an introduction to the cryptographic theory of random number generation. Here we give a detailed analysis of the QR-generator.

1 The QR-generator is pseudo-random.

Recall from Brands and Gill (1995) that the QR-generator is defined as follows: for suitably chosen integers x_0 and m, define

$$x_n = \begin{cases} x_{n-1}^2 \bmod m & \text{if } x_{n-1}^2 \bmod m < m/2; \\ m - (x_{n-1}^2 \bmod m) & \text{otherwise.} \end{cases} \tag{1}$$

and let

$$y_n = \text{lsb}(x_n)$$

denote the least significant bit of x_n. From the theoretical construction of Blum and Micali, it is clear that the proof of pseudo-randomness of the QR-generator is complete if it can be shown that the function displayed in (1) defines a one-way permutation under the plausible assumption that it is infeasible to factor Blum integers, and that the least significant bit y_n is hard-core for this permutation. We discuss the proofs of this. For further details, the reader is referred to the original articles of Ben-Or, Chor and Shamir (1983) and Alexi, Chor, Goldreich and Schnorr (1988).

First, we introduce some notions and results from elementary number theory that will be necessary to understand the proofs.

*Centre for Mathematics and Computer Science, Kruislaan 413, 1098 SJ Amsterdam, Netherlands.

†Mathematical Institute, University Utrecht, Budapestlaan 6, 3584 CD Utrecht, Netherlands.

1.1 Number theoretic preliminaries.

The subset of integers in $\mathbb{Z}_m$ that are co-prime with m is a multiplicative group, denoted by $\mathbb{Z}_m^*$. Let x be an element of $\mathbb{Z}_m^*$, for which there exists an element $y \in \mathbb{Z}_m^*$ such that $x = y^2 \bmod m$. Such an x is called a quadratic residue modulo m, and the set of all quadratic residues modulo m is denoted by QR_m.

If $x^2 \bmod m$ equals $y^2 \bmod m$, then $m|(x^2 - y^2)$ and hence $m|(x - y)(x + y)$. In case m is a prime number, this implies that $x = \pm y \bmod m$; a quadratic residue modulo a prime p has exactly two square roots. In particular, the square roots of 1 modulo p are 1 and $-1 \bmod p$. A further immediate consequence is that exactly half of the elements modulo p are quadratic residues.

It is well-known that $\mathbb{Z}_p^*$ is cyclic when p is a prime, which means that there exists an element $g \in \mathbb{Z}_p^*$ the powers of which exhaust all of $\mathbb{Z}_p^*$. Since $\mathbb{Z}_p^*$ contains $p - 1$ elements, the order of g equals $p - 1$. If a is even, then $g^{a/2} \bmod p$ is a square root of $g^a \bmod p$. Because exactly half of the elements in $\{0, \ldots, p - 2\}$ are even, the quadratic residues modulo p are precisely those elements that are an even power of g. For $x \in \mathbb{Z}_p^*$, the Legendre symbol (x/p) of x modulo p is defined to equal 1 if and only if x is a quadratic residue modulo p, otherwise it equals -1. We already mentioned that the square roots of 1 modulo p are -1 and 1. From Fermat's little theorem we hence have that $x^{(p-1)/2} = \pm 1 \bmod p$. In combination with the foregoing, it easily follows that $x^{(p-1)/2} = (x/p) \bmod p$, since each $x \in \mathbb{Z}_p^*$ can be written as $g^a \bmod p$ for some unique $a \in \{0, \ldots, p - 2\}$. This is called Euler's criterion. In particular, if the prime p is congruent to 3 modulo 4 then $(p - 1)/2$ is odd, and by Euler's criterion -1 is not a quadratic residue modulo p. This fact will turn out to be of crucial importance for the construction of the one-way permutation.

If m is the product of two distinct primes p and q, then from the Chinese Remainder theorem it follows that $\mathbb{Z}_m^*$ is isomorphic to $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$, with an isomorphism that maps $x \in \mathbb{Z}_m^*$ onto $(x \bmod p, x \bmod q)$. A particular consequence of this is that a number x is a quadratic residue modulo m if and only if it is a quadratic residue modulo both p and q. For $x \in \mathbb{Z}_m^*$, the Jacobi symbol (x/m) of x modulo m is defined to equal $(x/p)(x/q)$. Contrary to the Legendre symbol, x need not be a quadratic residue modulo m if its Jacobi symbol modulo m equals 1, since its Legendre symbols modulo both p and q may equal -1. As a matter of fact, a consequence of the isomorphism between $\mathbb{Z}_m^*$ and $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$ is that precisely half of the elements of $\mathbb{Z}_m^*$ have Jacobi symbol 1, and only half of these elements are quadratic residues. Since obviously none of the elements with Jacobi symbol -1 can be a quadratic residue modulo m, exactly one quarter of the elements of $\mathbb{Z}_m^*$ are quadratic residues. From a simple counting argument it follows that each quadratic residue modulo m has exactly four square roots modulo m.

1.2 The permutation.

Now consider the case that m is the product of two distinct odd primes p and q that are both congruent to 3 modulo 4, and that have approximately equal binary length. Composites of this form are called, as mentioned in Part I, Blum integers. We will denote the binary length of m by k. From the above discussion, it follows that the Jacobi symbol of -1 modulo m is 1, whereas -1 is not a quadratic residue modulo m. From this and the Chinese remainder theorem it follows that the outcome of the pair of Legendre symbols modulo both p and q of each of the four square

roots of $x^2 \bmod m$ is equal to precisely one of $(-1,-1)$, $(-1,1)$, $(1,-1)$ and $(1,1)$. In particular, exactly one of the four roots of a quadratic residue modulo a Blum integer is itself a quadratic residue.

In other words, the mapping $f_m : \mathrm{QR}_m \to \mathrm{QR}_m$ defined by

$$f_m(x) = x^2 \bmod m, \tag{2}$$

induces a permutation on QR_m. In order to formally apply the cryptographic theory that we described in part I, we in fact need to define the permutation such that m is included as an argument, i.e., $f(m,x) = (m, x^2 \bmod m)$. For simplicity, we will write m always as a subscript.

Although it is easy to prove (see Rabin, 1979) that the problem of inverting this permutation is probabilistic polynomial-time equivalent to factoring m (in a manner similar to the proof we will discuss shortly for a modification of (2)), the strongest known result (see Blum, Blum and Shub, 1986) for determining the least significant bit of its inverse is that it is as hard a task as solving the so-called Quadratic Residuacity problem. This problem consists of distinguishing between the quadratic residues and non-residues among the set of elements modulo a Blum integer that have Jacobi symbol equal to 1. Although solving this problem is believed to require factoring of the modulus, there is no proof known for this; assuming that the Quadratic Residuacity problem is infeasible seems a stronger assumption than assuming that Blum integers are infeasible to factor.

Since we are interested in a bit generator whose pseudo-randomness follows rigorously from the assumption that factoring is difficult, the definition of the permutation must be slightly modified. For this, we first define

$$\mathrm{abs}_m(z) = \begin{cases} z \bmod m & \text{if } z \bmod m < m/2; \\ m - (z \bmod m) & \text{otherwise.} \end{cases}$$

Redefine $f_m(\cdot)$ as

$$f_m(x) = \mathrm{abs}_m(x^2 \bmod m). \tag{3}$$

Clearly, iterating this function we get precisely the recursion (2) defining the QR-generator. If $x^2 \bmod m$ is not less than $m/2$, then $m - (x^2 \bmod m)$ is. Because the Legendre symbol, and hence also the Jacobi symbol, is multiplicative, it follows that $((m-y)/m) = (-y/m) = (-1/m)(y/m) = (y/m)$ if m is a Blum integer. Consequently, exactly one square root of a quadratic residue is less then $m/2$, since every quadratic residue modulo m has exactly two square roots with Jacobi symbol 1. Hence, the redefined $f_m(\cdot)$ also induces a permutation, this time on the set $\{x \in \mathbb{Z}_m^* \mid 0 \le x < m/2 \text{ and } (x/m) = 1\}$.

Just as for the original permutation (2), inverting $f_m(\cdot)$ is probabilistic polynomial-time equivalent to factoring the modulus. This can be seen as follows. On given as input a number $z = y^2 \bmod m$, such that $0 \le z < m/2$ and a square root y with Jacobi symbol -1 of z is known, an algorithm for inverting $f_m(\cdot)$ will output with probability lower-bounded by, say, $1/\delta(k)$ a square root x of z in the domain of $f_m(\cdot)$. Since $(x/m) = 1$, and we know that $(-1/m) = 1$, it cannot be the case that $x = \pm y \bmod m$. Hence, $pq|(x-y)(x+y)$ implies that exactly one of p, q divides $x+y$ evenly. Using the Euclidean gcd-algorithm, $\gcd(m, x+y)$ can be feasibly computed, thereby providing one (and hence both) of the prime factors of m. The probability

that a randomly selected $y \in \mathbb{Z}_m^*$ has Jacobi-symbol -1 is equal to $1/2$, as is the probability that $0 < z < m/2$. If y is uniformly distributed over the elements with Jacobi symbol -1, then z is uniformly distributed over QR_m. Define a Bernoulli experiment as consisting of generating at random an element $y \in \mathbb{Z}_m^*$, feeding $z = y^2 \bmod m$ to the inverting algorithm in case $z < m/2$ (otherwise, select a new y), and checking whether the output of the algorithm is a square root of z. Then one need only do independent repetitions of this Bernoulli experiment an expected number of $2\delta(k)$ times in order to retrieve a square root that enables factoring of m. If $\delta(\cdot)$ is a polynomial in k, and the inverting algorithm runs in polynomial time, then this procedure runs in expected polynomial time.

1.3 Sampling from the domain.

If we can prove that the least significant bit of $f_m(\cdot)$ is hard-core, we are finished with the proof that the QR-generator is pseudo-random assuming that Blum integers are infeasible to factor. Before turning to this issue, we verify the necessary conditions for one-way permutations mentioned earlier. Namely, the number of Blum integers of length k must grow exponentially with k, and pairs (m, x) must be feasibly samplable at random.

The Prime Number theorem states that the number of primes less than n is asymptotic to $n/\log n$. Furthermore, Dirichlet's theorem on primes in a progression states that the fraction of primes congruent to $a \bmod b$, with $\gcd(a, b) = 1$, asymptotically has constant density (namely, $1/\varphi(b)$) among the set of all primes. From this, it follows easily that there are indeed exponentially many (in k) Blum integers.

By performing independent Bernoulli experiments, randomly and independently picking in each experiment an integer of specified binary length and checking it for primality, the expected number of experiments needed to hit upon a prime congruent to 3 modulo 4 of a specified length is polynomial (in particular, linear) in the specified length. Since there exist well-known probabilistic algorithms for verifying primality in polynomial time, this shows that Blum integers can be feasibly sampled according to a uniform distribution by repeating this procedure twice.

The other part of the argument (seed) of the QR-generator is a randomly chosen quadratic residue in $\mathbb{Z}_m^*$. Because $\mathbb{Z}_m^*/\mathbb{Z}_m \to 1$ if $m \to \infty$, a randomly selected $y \in \{0,1\}^k$ has a very high probability of being in $\mathbb{Z}_m^*$. This can be checked in polynomial time using the Euclidean gcd-algorithm. Now set $x = y^2 \bmod m$ then by elementary probability theory, x has been chosen from QR_m with uniform probability.

It is amusing that in a theory which depends on the notion of a probabilistic polynomial-time algorithm to characterise feasible and infeasible problems, one should go to so much trouble to describe how randomness can be generated, or rather expanded, in a deterministic way. A probabilistic algorithm is supposed to be able to generate its own fair coin tosses, so looking from inside the theory, random number generators are not needed; they already exist!

1.4 The hard-core bit.

The hardest and most lengthy part of the proof is to show that the existence of a feasible algorithm that can guess the least significant bit of $f_m^{-1}(\cdot)$ with a probability of success significantly

exceeding 1/2 can be used to construct a feasible algorithm for inverting $f_m^{-1}(\cdot)$. Together with the result of Subsection 1.2, such a polynomial-time reduction implies that the least significant bit of $f_m(\cdot)$ is essentially as hard to determine as factoring the modulus. As required by the general construction of Blum and Micali for pseudo-random bit generators, the reduction must work even if the success probability of the algorithm only 'slightly' exceeds 1/2. We will henceforth denote by $\mathcal{O}_{\text{lsb}}$ an 'oracle' that, on given as inputs a Blum integer m and a number x in the range of $f_m(\cdot)$, outputs a guess for the least significant bit of $f_m^{-1}(x)$ that is correct with probability at least $1/2 + 1/k^c$ for some constant c.

The main idea to construct a feasible algorithm that inverts $f_m(\cdot)$ by calling $\mathcal{O}_{\text{lsb}}$ at most polynomially many times is to retrieve the inverse of $f_m(\cdot)$ by using a greatest common divisor algorithm. To this end, two randomly chosen multiples of $f_m(x)$ are computed, $f_m(ax \bmod m)$ and $f_m(bx \bmod m)$, and an attempt is made to compute the greatest common divisor of $[ax]_m$ and $[bx]_m$ by manipulating the 'permuted values' $f_m(ax \bmod m)$ and $f_m(bx \bmod m)$. Let us call this an 'experiment'. Here, $[y]_m$ denotes the value congruent modulo m to y such that $-m/2 < [y]_m < m/2$. Note that $y \bmod m$ and $[y]_m$ are related in the following way:

$$[y]_m = \begin{cases} y \bmod m & \text{if } y \bmod m < m/2; \\ (y \bmod m) - m & \text{otherwise.} \end{cases}$$

Furthermore, $f_m([y]_m) = f_m(y \bmod m)$ and, denoting absolute value by $|\cdot|$, $|[y]_m| = \text{abs}_m(y)$.

When the gcd-algorithm has finished, a representation of $\gcd([ax]_m, [bx]_m)$ in the form $[dx]_m$ should be known, such that d and $f_m(dx \bmod m)$ $(= f_m([dx]_m))$ are known. If $[ax]_m$ and $[bx]_m$ are co-prime then $[dx]_m = \pm 1$, and so $f_m(dx) = 1$. Although $f_m(dx \bmod m) = 1$ does not necessarily imply that $[dx]_m = \pm 1$ (since 1 has four square roots modulo m), we can always check whether this is the case by comparing $f_m(\pm d^{-1} \bmod m)$ to $f_m(x)$. If these values are equal, then we have a good chance that one of $d^{-1} \bmod m$, $-d^{-1} \bmod m$ is in the domain of $f_m(\cdot)$ and we are finished. (Computation of inverses modulo m can be done in polynomial time using the Extended Euclidean algorithm; the prime factorisation of m need not be known for this.) Otherwise, we perform a new experiment, i.e., select new a and b (independently) and start over again.

We are only interested in whether the outcome of an experiment is or is not equal to x, so this is in fact a Bernoulli experiment. Suppose for the moment that we are indeed able to construct the above inverting algorithm. By a theorem of Dirichlet, the probability that two randomly chosen integers less than m are co-prime tends to a constant (namely $6/\pi^2$) as m tends to infinity. Hence, the Bernoulli experiment of selecting two random multiples of $f_m(x)$ and running the gcd-algorithm has a probability of retrieving x that equals a constant fraction of the probability that the gcd-algorithm outputs the correct answer. Consequently, if the probability that the gcd-algorithm outputs the correct answer can be lower-bounded by the inverse of some polynomial in k, then the experiment need only be repeated an expected number of times that is polynomial in k in order to retrieve x. As we will show, this expected running time will be blown up by another polynomial factor due to the fact that we will have to run polynomially many copies of the gcd-algorithm for each experiment, assuming one out of polynomially many possibilities in each run. This still keeps the final running time polynomial!

Therefore, we can invert $f_m(x)$ if a polynomial-time algorithm can be constructed that implements the gcd-algorithm, using only $\mathcal{O}_{\text{lsb}}$. The Euclidean gcd-algorithm makes repeated use of testing whether one integer is greater than the other. However, this algorithm cannot feasibly be implemented by using only the permuted values, since from $f_m(ax \bmod m)$ and $f_m(bx \bmod m)$ one cannot in general determine whether (the absolute value of) $[ax]_m$ exceeds $[bx]_m$; in fact, a feasible algorithm that can do such a test without erring can be simply converted into one that factors m.

1.4.1 Constructing the gcd-algorithm.

What is needed is a gcd-algorithm that only has to make decisions based on the least significant bits of the involved integers, since we have access to $\mathcal{O}_{\text{lsb}}$. This can be realised by using a binary gcd-algorithm such as the Brent-Kung gcd-algorithm, although unfortunately the way this algorithm (or any other known one) must make calls to $\mathcal{O}_{\text{lsb}}$ turns out to be far from straightforward.

The Brent-Kung algorithm makes use of the following three tests. Firstly, if A and B are two integers, not necessarily positive, and $|A|$ and $|B|$ are both even, then $\gcd(A, B) = 2\gcd(A/2, B/2)$. Secondly, if $|A|$ is odd and $|B|$ even then $\gcd(A, B) = \gcd(A, B/2)$. Thirdly, if both $|A|$ and $|B|$ are odd, then $\gcd(A, B) = \gcd(A, (A + B)/2) = \gcd(A, (A - B)/2)$, and the absolute value of exactly one of $(A + B)/2$, $(A - B)/2$ is even again. We assign that value to B. Because of the third test, we can iterate. If necessary we must swap A and B before we enter the third test to make sure that the absolute value of B exceeds that of A. In that way, we are guaranteed to keep on making progress and finally obtain the greatest common divisor in A. As a matter of fact, we might as well test whether A and B should be swapped by comparing their binary lengths $\text{length}(A)$ and $\text{length}(B)$. This turns out to be crucial for successful implementation of the 'permuted' version of the gcd-algorithm.

The precise description of the Brent-Kung algorithm is as follows. Given two integers A and B such that $\text{length}(A), \text{length}(B) \leq k$ and A odd, repeat the following steps until $B = 0$:

Step 1. While $\text{lsb}(|B|) = 0$ do $B \leftarrow B/2$; $\text{length}(B) \leftarrow \text{length}(B) - 1$ end.

Step 2. If $\text{length}(B) < \text{length}(A)$ then swap(A, B); swap($\text{length}(A), \text{length}$(B)) end.

Step 3. If $\text{lsb}(|(A + B)/2|) = 0$ then $B \leftarrow (A + B)/2$ else $B \leftarrow (A - B)/2$ end.

When the algorithm halts ($B = 0$), the variable A contains the greatest common divisor we were looking for. A simple counting argument reveals that no more than $6k + 3$ evaluations of $\text{lsb}(\cdot)$ are needed until B becomes zero. Notice that if A is not odd, then $\text{lsb}(|(A + B)/2|)$ in step 3 makes no sense; this is not a problem since we can reduce A beforehand to make it odd.

With $A = [ax]_m$ and $B = [bx]_m$, this algorithm can be implemented while working only with $f_m(ax \bmod m)$ and $f_m(bx \bmod m)$, given access to $\mathcal{O}_{\text{lsb}}$. We first introduce the notation $\text{par}_m(\cdot)$ for $\text{lsb}(\text{abs}_m(\cdot))$. Note that

$$\text{lsb}(|B|) = \text{lsb}(|[bx]_m|) = \text{par}_m(bx \bmod m).$$

In order to obtain $\text{lsb}(|B|)$ we must hence build an algorithm that computes $\text{par}_m(\cdot)$ for elements in $\mathbb{Z}_m^*$, using $\mathcal{O}_{\text{lsb}}$.

Secondly, note that if $|B|$ is even, then $B/2$ (which is equal to $[bx]_m/2$) may be computed as $[(2^{-1}b \bmod m)x]_m$. This follows from the fact that one can apply the modulo m operator on intermediate results. Although we never know B explicitly since we do not know x, we can keep track of B by keeping track of b. Furthermore, length(B) can be kept track of by diminishing a counter each time B is (implicitly) divided by 2 in Step 1. In that way, we can also make sure that we know when to (implicitly) swap A and B.

Of course, similar relations can be applied for Step 3 of the gcd-algorithm; we can implicitly compute $(A \pm B)/2$ as $[(2^{-1}(a \pm b) \bmod m)x]_m$, and hence keep track of this value by working with $2^{-1}(a \pm b) \bmod m$.

We are now prepared to describe in detail how the inverting algorithm works, based on the permuted gcd-algorithm. Suppose we are given a Blum integer m and $f_m(x)$ for some unknown x in the domain of $f_m(\cdot)$ that is to be determined. Generate independently at random two elements $a, b \in \mathbb{Z}_m^*$, and set length(A) = length$(B) = k - 1$. For the moment, just assume that A is odd (we will return to this shortly). Repeat the following steps until $b = 0$:

Step 1. While $\text{par}_m(bx \bmod m) = 0$ do $b \leftarrow 2^{-1}b \bmod m$; length$(B) \leftarrow$ length$(B) - 1$ end.

Step 2. If length$(B) <$ length(A) then swap(a, b); swap(length(A), length(B)) end.

Step 3. If $\text{par}_m(2^{-1}(a+b)x \bmod m) = 0$ then $b \leftarrow 2^{-1}(a+b) \bmod m$; else $b \leftarrow 2^{-1}(a-b) \bmod m$ end.

If all calls to $\text{par}_m(\cdot)$ were answered correctly (otherwise, the procedure might not halt in polynomial time), then in the end A contains the greatest common divisor. Although we only know A implicitly, we have kept track of a. Due to the fact that squaring modulo m is a multiplicative function, we can compute $f_m(A)$:

$$f_m(A) = f_m(ax \bmod m) = \text{abs}_m((a^2 \bmod m) f_m(x) \bmod m),$$

since we know $f_m(x)$. As mentioned earlier, if $f_m(A) = 1$ then with high probability $A = \pm 1$. We already explained how to determine if $\pm a^{-1} \bmod m$ equals x. If we were unsuccessful, we choose new $a, b \in \mathbb{Z}_m^*$ and do another experiment.

It remains to build a feasible algorithm for determining $\text{par}_m(\cdot)$ for elements in $\mathbb{Z}_m^*$ from $\mathcal{O}_{\text{lsb}}$. Although we do not explicitly know the numbers we want to know $\text{par}_m(\cdot)$ of, since they are all multiples of the unknown number x, we do have some information about them. Namely, we kept track of a and b. As shown above, $\text{lsb}(|B|) = \text{par}_m(bx \bmod m)$, and we know b. Likewise, $\text{lsb}(|(A+B)/2|) = \text{par}_m(2^{-1}(a+b)x \bmod m)$. In other words, all requests for $\text{par}_m(\cdot)$ are of the form $\text{par}_m(dx \bmod m)$ such that $d \in \mathbb{Z}_m^*$ is known. We will show shortly how this can be put to use. An important thing to note is that $\text{abs}_m(dx \bmod m)$ gets smaller all the time, since $|A|$ and $|B|$ decrease as the gcd-algorithm progresses. That is, if the values $|A| = \text{abs}_m(ax \bmod m)$ and $|B| = \text{abs}_m(bx \bmod m)$ are 'small' to begin with, then all values $dx \bmod m$ for which $\text{par}_m(\cdot)$ is requested are also small in this sense (unless the parity algorithm makes mistakes!).

To make sure that an experiment, in which we attempt to retrieve the greatest common divisor, will not loop more than a polynomial number of steps (which can happen if the bit determined

for $\text{par}_m(\cdot)$ is incorrect), the number of times that $\text{par}_m(\cdot)$ is requested must be kept track of in a counter. If the counter exceeds the strict upper bound ($6k-3$ in this case), then something went wrong and the specific run of the experiment can be halted.

Finally, we mentioned that A must be odd. This is simple to resolve: either we do not care whether A is odd (since this only happens only with probability 1/2) and then the 'upper-bound' mechanism takes care of it, or we test $\text{par}_m(ax \bmod m)$ in advance using the same mechanism we need anyway.

1.4.2 Constructing the parity algorithm.

The basic idea of the construction of the parity algorithm is to infer $\text{par}_m(dx \bmod n)$ by comparing the least significant bits of $s + dx \bmod m$ and s, for randomly chosen $s \in \mathbb{Z}^*_{\lfloor m/2 \rfloor}$. Only if no 'wraparound' 0 occurs when s is added to dx, then

$$\text{par}_m(dx \bmod n) = \text{lsb}(s) \oplus \text{lsb}(s + dx \bmod m). \tag{4}$$

This relation will allow us to compute $\text{par}_m(\cdot)$ using $\mathcal{O}_{\text{lsb}}$. To ensure that wraparounds have low probability of occurrence, we define $dx \bmod m$ to be 'small' if and only if $\text{abs}_m(dx \bmod m) < m/\delta(k)$ for some polynomial $\delta(\cdot)$. If $dx \bmod m$ is small, then the probability of a wraparound 0 is less than $1/\delta(k)$ if s is uniformly distributed. We will return to this in detail when we discuss the potential sources of errors.

Now, A and B are not known explicitly, and so for randomly chosen s the values of $s + dx \bmod m$ (and in particular their least significant bits) are all unknown as well. However, we have access to $\mathcal{O}_{\text{lsb}}$, which outputs (with some advantage over guessing) least significant bits of arbitrary numbers that are in the domain (!) of $f_m(\cdot)$, when given their permuted values. Hence, we can still be successful if there is a way to compute the permuted values $f_m(s)$ and $f_m(s + dx \bmod m)$ of s resp. $s + dx \bmod m$, for randomly chosen s. This is where the fact that we always know d comes in. Namely,

$$f_m(dx \bmod m) = \text{abs}_m((d^2 \bmod m) f_m(x)),$$

which can be computed since we know $f_m(x)$.

There is one complication here: in contrast to the method for generating 'multiplicative randomisations', such as $f_m(dx \bmod m)$, it is not known how to feasibly generate 'additive randomisations' such as $f_m(s + dx \bmod m)$ for which s itself is known. Indeed, a feasible algorithm for doing so would be a major step towards a feasible algorithm for factoring. Therefore, we use the fact that

$$f_m(rx + dx \bmod m) = \text{abs}_m(f_m(r + d \bmod m) f_m(x) \bmod m),$$

so $rx \bmod m$ can play the role of s. Now the complication is obvious: $s = rx \bmod m$ itself is not known, and so when using (4) we must ask $\mathcal{O}_{\text{lsb}}$ also for $\text{lsb}(s)$. This causes the undesirable effect of 'error-doubling' when querying $\mathcal{O}_{\text{lsb}}$. In fact, it is not even known whether s generated in this way is less than $m/2$ (again, a feasible algorithm for determining this would lead to a

simple algorithm for factoring!) which, as we already mentioned (albeit without explanation), must be the case.

From this discussion it will be clear that there are various potential sources for errors when trying to apply (4) using $\mathcal{O}_{\text{lsb}}$. We next consider these in detail, in order to determine a strategy for controlling them.

1.4.3 Potential sources of errors.

At this point, the reader may want to skip over to Subsection 1.5, if he is not concerned with the gory details of the error-sources and how to control them.

Let us consider what happens when we try to compute $\text{par}_m(\cdot)$ based on (4) by using $\mathcal{O}_{\text{lsb}}$. We feed $\mathcal{O}_{\text{lsb}}$ with $f_m(s \bmod m)$ and $f_m(s + dx \bmod m)$, and take the exclusive-or of the two outputs as a guess for $\text{par}_m(dx \bmod m)$. This is called a dx-measurement. The correctness of a dx-measurement cannot feasibly be verified. Nevertheless, if a dx-measurement would be correct with probability that exceeds $1/2$ by some fraction that is lower-bounded by the inverse of some polynomial in k, then one can perform a great many (say $j(k)$ for some polynomial $j(\cdot)$) dx-measurements by choosing r independently at random in each dx-measurement. Considering the majority value obtained in all dx-measurements as a final guess for $\text{par}_m(dx \bmod m)$, the probability that the final guess is correct can then be lower-bounded if the measurements are all independent: the Chernoff bound gives a lower bound of $1 - O(1/2^{j(k)})$. That is, the calls for $\text{par}_m(\cdot)$ in an experiment are returned correctly with probability almost 1.

However, the oracle $\mathcal{O}_{\text{lsb}}$ itself is allowed to err with probability close to $1/2$. So, even if the error probability of the oracle would be the only source of errors in a dx-measurement, then the probability that the exclusive-or is not equal to the true least significant bit of $dx \bmod m$ can still be almost twice the error probability of $\mathcal{O}_{\text{lsb}}$. Since we query $\mathcal{O}_{\text{lsb}}$ for two numbers in a dx-measurement, we need an error-probability for $\text{par}_m(dx \bmod m)$ that exceeds $1/2$ by a significant fraction in order to make sure that the majority outcome is almost always correct.

To overcome this problem, the dx-measurement procedure is modified such that the least significant bit of s is known beforehand with at least 'significant' probability. One can then query $\mathcal{O}_{\text{lsb}}$ in all $j(k)$ trials for $\text{lsb}(s_i + dx \bmod m)$, $1 \leq i \leq j(k)$, only. The error probability in a dx-measurement can then be approximated by the error probability of $\mathcal{O}_{\text{lsb}}$, and we may realistically hope that the probability of error in a single dx-measurement still significantly exceeds 1/2.

Before we show how to do this, we need to be aware that there are two additional potential sources of errors in dx-measurements, other than the errors made by the oracle. We already mentioned that a wraparound 0 may occur, in which case the right-hand side of (4) does not equal $\text{par}_m(dx \bmod m)$, and that this happens with probability less than $1/\delta(|m|)$ if $dx \bmod m$ is small. We also noticed earlier that if $ax \bmod m$ and $bx \bmod m$ are both small to begin with, then all the values that are assigned in the gcd-algorithm to $ax \bmod m$ and $bx \bmod m$ (through their permuted values) are small as well (i.e., the values $dx \bmod m$ for which the parity is requested, are all small) if par_m is always returned correctly. Since $\delta(\cdot)$ is a polynomial, the probability that $ax \bmod m$ and $bx \bmod m$ are both small to begin with is lower-bounded by the inverse of some polynomial in k, so one out of an expected number of polynomially many experiments contains such $ax \bmod m$ and $bx \bmod m$. This very important detail enables us to easily take care of this error source: by

performing polynomially many independent experiments, we only need an expected number of experiments that is polynomial in k in order to have two values a and b for which $ax \bmod m$ and $bx \bmod m$ are both small. In that particular experiment, this type of error has a very small probability, if only we make sure that s is uniformly distributed over $0 \leq s < m/2$ instead of over $\mathbb{Z}_m^*$. Notice that there is some peculiar interaction here: all the values for which $\text{par}_m(\cdot)$ is requested will only get smaller throughout the gcd-algorithm if $\text{par}_m(\cdot)$ is (with overwhelming probability) correctly computed each time, and vice versa the probability that a wraparound 0 occurs is small if the value for which $\text{par}_m(\cdot)$ is requested is small.

A second source of potential errors was also drawn attention to before: the success probability of $\mathcal{O}_{\text{lsb}}$ exceeds 1/2 by some 'significant' fraction *only on the domain* of $f_m(\cdot)$. If the oracle is queried for $\text{lsb}(s + dx \bmod m)$ (recall that in the revised procedure we just announced for sampling the oracle, we will no longer query the oracle for $\text{lsb}(s)$), the bit that the oracle outputs does not necessarily help us: if $s + dx \bmod m$ is not in the domain of $f_m(\cdot)$, then the outcome of $\mathcal{O}_{\text{lsb}}$ corresponds to the inverse of $f_m(s + dx \bmod m)$, a value that does not equal $s + dx \bmod m$. There are two situations in which this may happen: the first is if $s + dx \bmod m$ has Jacobi symbol modulo m equal to -1, the second if $s + dx \bmod m$ is greater than $m/2$. The first situation is easy to detect without knowing x, since $(s + dx/m) = (r + d/m)(x/m) = (r + d/m)$ because $s = rx \bmod m$ and $(x/m) = 1$. (Although from the definition of the Jacobi symbol it seems that one must be able to factor in order to compute Jacobi symbols modulo m, the Quadratic Reciprocity theorem of Gauss provides a feasible algorithm for this task without ever using the factoring of m.) If this happens, $\mathcal{O}_{\text{lsb}}$ will not be queried but the flip of a fair coin will be used as a guess for $\text{par}_m(dx \bmod m)$. In this situation, our guess is correct with probability exactly 1/2. The second situation happens with very small probability if $dx \bmod m$ is small, which we can take care of by choosing the polynomial $\delta(\cdot)$ appropriately large and $s < m/2$ (exactly as in the first potential source of errors).

1.4.4 Implementing the dx-measurements.

As mentioned, to overcome error-doubling the $j(k)$ points $s_i = r_i x \bmod m$, $1 \leq i \leq j(k)$, with known least significant bit are generated according to the following procedure. Generate independently at random two elements u, v of $\mathbb{Z}_m^*$, and denote $ux \bmod m$ by y and $vx \bmod m$ by z. Although y and z are not known, all possibilities for their least significant bits can be tried, as well as their location in one of the intervals $[(i/j(k)^c)m, (i + 1/j(k)^c)m)$ for $0 \leq i < j(k)^c$ for some suitable constant $c > 1$. In total, there are $4j(k)^{2c}$ possibilities that have to be considered, of which only one is correct. For $1 \leq i \leq j(k)$, define $s_i = r_i x \bmod m$ to equal $\text{abs}_m(y + iz \bmod m)$. Then $s_i < m/2$ as we required. Since y and z are known up to $m/j(k)^c$, $y + iz \bmod m$ is known up to at most $2m/j(k)^{c-1}$. The probability that $y + iz \bmod m$ does not fall in an interval of length $2m/j(k)^c$ around 0 or $m/2$ is $4/j(k)^{c-1}$ (since it is uniformly distributed over $\mathbb{Z}_m$), and in that case the least significant bit of s_i can be computed correctly from the least significant bits of y and z, and the value of i.

Although it is not known what the location of y and z is, nor what their least significant bits are, the fact that there are only polynomially many possibilities implies that there are only polynomially many possible values that the 'vector' $(s_1, \ldots, s_{j(k)})$ can take on: each guess for the

location of y and z and their least significant bits, specifies exactly one such possibility. Hence, each of the values that the vector $(s_1, \ldots, s_{j(k)})$ can take on can be tried for when doing the $j(k)$ dx-measurements.

Assume for the moment that we are dealing with the possibility in which the values we assigned to $s_1, \ldots, s_{j(k)}$ are indeed the correct ones. Taking all the error sources into account, it can easily be shown that the total error probability in each one of all $j(k)$ dx-measurements in which these correct least significant bits for s_i are used, is still greater than 1/2 by a fraction that is lower-bounded by the inverse of some polynomial in k (i.e., it is still 'significantly greater than 1/2), if only $dx \bmod m$ is small. Although the $j(k)$ points s_i generated in this way are not mutually independent (if they were, then we would have to enumerate *exponentially* many possibilities for the least significant bits, which would not be sufficient to prove the desired result!), and hence the Chernoff bound cannot be used, it is not hard to show that they are pairwise independent. Therefore we can still apply Chebyshev's inequality to lower-bound the probability of correctness of the majority vote by $1 - O(1/j(k))$.

The way we use this for the inverting algorithm is as follows. In one experiment (i.e., using one pair of values for (a, b)), $\text{par}_m(dx \bmod m)$ must be evaluated for polynomially many values of d. For each of these polynomially many calls for $\text{par}_m(\cdot)$, we generate u, v at random, we can enumerate polynomially many possible values for the vector $(s_1, \ldots, s_{j(k)})$ that we can derive from u, v, and use these s_i values in dx-measurements. In each of these polynomially many situations, the majority outcome of the dx-measurements is taken as a guess for $\text{par}_m(\cdot)$. We know for sure that in one of all these situations we assumed the correct values for each of the s_i; in that particular case, the majority vote of the dx-measurements is a guess for $\text{par}_m(\cdot)$ that is correct with probability close to 1. In all the other copies, the returned guess for $\text{par}_m(\cdot)$ might be wrong with considerable probability, and those runs of the experiment might not halt in polynomial time; however, this is taken care of by the strict upper-bound mechanism. In essence, we are running polynomially (for each of the calls to $\text{par}_m(\cdot)$) times polynomially (for each of the possible values of the vector) many copies of each experiment, which is a polynomial number of copies. Since we already showed that only an expected number of polynomially many experiments is needed to find the inverse, the proof is completed.

1.5 Applying the construction of Blum and Micali.

By using the general construction of Blum and Micali that we described in Part I, taking as a seed a random element $(m, x_0 \in QR_m)$ with m in the set of Blum integers, one obtains a bit generator which at the n-th step outputs the least significant bit of the second argument of $f^n(m, x)$, where

$$f(m, x) = \begin{cases} (m, x^2 \bmod m) & \text{if } x^2 \bmod m < m/2; \\ (m, m - (x^2 \bmod m)) & \text{otherwise.} \end{cases}$$

If the problem of factoring Blum integers is infeasible, then the generator passes all polynomial-time statistical tests. By combining the bits in groups of suitable size, and considering this as the binary representation of an integer, one can construct number generators.

Alternatively, the QR-generator can be defined by the recurrence

$$x_n = x_{n-1}^2 \bmod m,$$

such that the bits

$$y_n = \text{lsb}(\text{abs}_m(x_n))$$

are output. That is, we compute the sequence x_n, and at each stage extract either the least significant bit of x_n (in case $x_n < m/2$) or complement it (if $x_n > m/2$). Clearly, for the same seed this produces exactly the same sequence of bits.

2 Further developments.

In this section we briefly touch upon further developments of the cryptographic theory.

2.1 Hard-core predicates of multiple bits.

A possible handicap in practice of generators based on the construction of Blum and Micali is that only one bit (the hard-core bit) is output per iteration of the one-way permutation. One would like to be able to extract at each stage *more* bits than just one in order to increase the efficiency.

The first idea that comes to mind is to try to come up with other (hard-core bits) of the permutation $f(\cdot)$ under consideration that are hard-core as well. For example, for the permutation defining the QR-generator, it can be shown that each of the bits in the $j(k)$ least significant positions is hard-core, if $j(k) = O(\log k)$. However, it would be wrong to output at each iteration all the bits that have *individually* been proven hard-core, because besides having the information $f(x)$, which by itself is not enough to compute a certain given hard-core bits of x, extra information is released (namely other hard-core bits).

As an extreme example one can think of a one-way permutation defined on a domain consisting entirely of elements for which both the most and the least significant bit have equal parity. Although these bits may be proven individually hard-core, revealing the least significant bit makes the most significant bit completely predictable. In the output sequence of the generator obtained when applying the construction of Blum and Micali this is readily apparent, since it consists of a stream of bits that are pairwise the same.

Hence, a notion of *simultaneously hard-core* bits is needed. This notion informally amounts to the requirement that no feasible algorithm can guess with significant probability of success any of the hard-core bits of the argument x of the one-way permutation $f(\cdot)$ even when, in addition to $f(x)$, it is given all the other hard-core bits of $f(x)$. This requirement in fact follows straightforwardly from the conditions that must obviously be satisfied in order to mathematically justify outputting all these bits at each iteration in the Blum-Micali construction. By extending the proof techniques sketched in the previous section, it has been shown in Alexi, Chor, Goldreich and Schnorr (1988) that the least $O(\log k)$ bits of the function defining the QR-generator in fact are simultaneously hard-core, and hence they can all be output per iteration of the permutation. The resulting generator is still pseudo-random under the assumption that it is infeasible to factor Blum integers.

2.2 Weakening the theoretical assumptions.

While the existence of a one-way permutation in the Blum-Micali construction is sufficient for the existence of pseudo-random bit generators, it is not necessary. Weaker conditions were given by Yao (1982), who showed in a constructive manner that it is sufficient to assume the existence of a one-way function; it need not be a permutation. In fact, it is sufficient if the function is 'somewhat' one-way, meaning that any feasible algorithm that tries to find an inverse of elements in the range of the function has a probability of failure that is lower-bounded by the inverse of some polynomial in the length of the input. That is, it must have some probability 'significantly' bigger than zero of failing to find an inverse, rather than an overwhelming probability. The idea is to transform any somewhat one-way function into a strongly one-way function, which in turn can be transformed into a more complicated one-way permutation that has a hard-core predicate. This permutation can be used in the Blum-Micali construction, which gives the desired result.

The first result to state necessary as well as sufficient conditions is due to Levin (1985). It is based on the existence of a special subclass of all one-way functions, those that are *one-way on iterates*. Intuitively, this condition ensures that the successively induced ensembles of strings in the range of the function do not 'degenerate' too soon. Using a special concept that he called isolation, Levin proved that there exists a pseudo-random bit generator if and only if there exists a function that is one-way on iterates. This theorem has the drawback that it seems quite difficult to test the plausibility of the assumption that $f(\cdot)$ is one-way on iterates. Also, it is not satisfactory in the sense that it is not clear if the existence of such special one-way functions is equivalent to the existence of ordinary one-way functions.

A step closer in this direction was made by the introduction by Goldreich, Krawczyk and Luby (1988) of another, larger subclass of all one-way functions, called *regular* one-way functions. This condition comes down to the requirement that all elements in the range of $f(\cdot)$ have the same number of preimages, and enabled them to prove that if there exists a regular function that is one-way, then there exists a pseudo-random bit generator. The main advantage of the notion of regularity is that many quite naturally arising functions are regular, so if any one of these can be shown to be one-way (under some plausible assumption) then this result can be used to construct a pseudo-random bit generator. The proof of this theorem is based on a feasible method to transform regular functions into functions that are one-way on iterates, after which Levin's theorem can be applied.

The most recent theoretic results in this area have shown that all of the above notions are in fact equivalent. Namely, a theorem of Impaggliazzo, Levin, Luby and Hastad states that there exists a one-way function if and only if there exists a pseudo-random generator. Actually, this was shown to hold in the *non-uniform* model by Impaggliazzo, Levin and Luby (1989), and later in the *uniform* model by Hastad (1990). (The difference between uniform and non-uniform complexity is something that we have deliberately not touched upon, since we expect the readers of this article to be mainly statistically oriented. The relevance of this difference only derives from the details of the proofs of certain theoretical constructions.) The proof techniques of Impaggliazzo, Levin, Luby and Hastad are quite complicated and use entirely different concepts (such as entropy) than those used to prove the earlier results. An open problem is whether the construction given in the proof of this theorem can be made efficient for practical use (i.e., a

low-degree polynomial upper-bound for the running times of the generator). The construction applied in the proofs results in a generator that has a running time bounded from below by a high degree polynomial; in practice it seems that the generators arising from the construction of Blum and Micali are the most efficient ones. The most efficient of all the known generators based on the construction of Blum and Micali is the QR-generator, which we described in the previous section.

3 Conclusion.

Although pseudo-randomness of the QR-generator arising from the cryptographic theory is based on a highly respectable assumption, it is likely that it performs better with respect to the standard and stringent statistical tests than the classical statistical methods, even if the assumption turns out not to be true. This is due to the fact that a statistical test that is not passed by the QR-generator will probably be extremely non-elementary (as the proof of pseudo-randomness for the QR-generator suggests). As a consequence of this, one can even expect the cryptographic methods to do better in this respect if (almost) all of the bits are output at each stage, instead of just the bits that are simultaneously hard-core (even though the thus arising generators are perhaps no longer pseudo-random in the theoretical sense).

In fact, although the important results of the cryptographic generators are asymptotic, one can expect the cryptographic generators to do better even when small parameter choices are involved. A detailed (and tedious) examination of the proofs of the theoretical results and the QR-generator reveals that one can get some definite statements such as 'If Blum integers with k bits cannot be factored using less than n_1 bit operations then no bit of the QR-generator with a modulus of k bits can be predicted with any algorithm using less than n_2 bit operations', for a value of n_2 depending on n_1. All the above obviously holds as well for other generators that have been proposed in the cryptographic literature.

It is interesting to investigate how many bits of the integers produced at each iteration of the cryptographic generators can be output such that all the standard and stringent statistical tests are passed, and what length of the parameters of the cryptographic generators is necessary to ensure that all standard and stringent statistical tests are passed. In particular, it is interesting to examine what the performance is of for example the QR-generator if a small modulus is chosen. A small amount of practical experience with the QR-generator (Brands, 1991), suggests that it is certainly as good, in the traditional sense of passing traditional statistical tests of randomness, as a linear congruential generator of similar size of which the most significant bit is output at each iteration.

4 Bibliography.

W. Alexi, B. Chor, O. Goldreich, and C.P. Schnorr (1988), RSA and Rabin functions: certain parts are as hard as the whole, *SIAM J. Comp.* **17**, 194–209.

M. Ben-Or, B. Chor, and A. Shamir (1983), On the cryptographic security of single RSA bits, *Proc. 15th ACM Symp. Theor. Comp.*, 421–430.

L. Blum, M. Blum, and M. Shub (1986), A simple unpredictable pseudorandom number generator, *SIAM J. Comp.* **15**, 364–383.

S.A. Brands (1991), *The Cryptographic Approach to Pseudorandom Bit Generation*, Master's thesis, Dept. Math., Univ. Utrecht.

S.A. Brands and R.D. Gill (1995), Cryptography, Statistics and Pseudo-Randomness (Part II), *Probability and Mathematical Statistics* **??**, ???–???.

O. Goldreich, H. Krawczyk, and M. Luby (1988), On the existence of pseudorandom generators, *Proc. 29th Ann. Conf. on Found. Comp. Science, IEEE*, 12–24.

J. Hastad (1990), Pseudorandom generators under uniform assumptions, *Proc. 22nd ACM Symp. on Theory of Comp.*, 395–404.

R. Impaggliazzo, L. Levin, and M. Luby (1989), Pseudorandom number generation from one-way functions, *Proc. 21nd ACM Symp. on Theory of Comp.*, 12–24.

L. Levin (1985), One-way functions and pseudorandom generators, *Combinatorica* **7**, 357–363.

M. Rabin (1979), Digitalized signatures and public key functions as intractable as factorization, Tech. Rep. 212, Lab. Comp. Sci., MIT.

A.C. Yao (1982), Theory and applications of trapdoor functions, *Proc. 23rd IEEE Symp. Found. Comp. Sci.*, 458–463.