Kryptoprojekt

Pseudo Random Bit Generation

December 2007

Katrine Mollerup Johansen, 20042707

Lene Mejlby, 20051795

1 Table of Contents

Katrine Mollerup Johansen, 20042707
Lene Mejlby, 20051795

1. Randomness

The word random is used to express lack of order, purpose, cause, or predictability in non-scientific parlance. A random process is a repeating process whose outcomes follow no describable deterministic pattern, but follow a probability distribution.

The term randomness is often used in statistics to signify well defined statistical properties, such as lack of bias or correlation.

1.1 Randomness versus unpredictability

Randomness is an objective property. Nevertheless, what *appears* random to one observer may not appear random to another observer. Consider two observers of a sequence of bits, only one of whom has the cryptographic key needed to turn the sequence of bits into a readable message. The message is not random, but is unpredictable for one of the observers. One of the intriguing aspects of random processes is that it is hard to know whether the process is truly random. The observer can always suspect that there is some "key" that unlocks the message. This is one of the foundations of superstition and is also what is a driving motive, curiosity, for discovery in science and mathematics.

Randomness can often be precisely characterized in terms of probability or expected value. For instance, though we cannot predict the outcome of a single toss of a fair coin, we can characterize its general behavior by saying that if a large number of tosses are made, roughly half of them will show up "Heads".

1.2 Generating randomness

The ball in a roulette can be used as a source of apparent randomness, because its behavior is very sensitive to the initial conditions.

It is generally accepted that there exist three mechanisms responsible for (apparently) *random* behavior in systems :

1. *Randomness* coming from the environment (for example hardware random number generators)

2. *Randomness* coming from the initial conditions. This aspect is studied by chaos theory, and is observed in systems whose behavior is very sensitive to small variations in initial conditions (such as pachinko machines, dice ...).
3. *Randomness* intrinsically generated by the system. This is also called **pseudorandomness**, and is the kind used in pseudo-random number generators. There are many algorithms (based on arithmetics or cellular automaton) to generate pseudorandom numbers. The behavior of the system can be determined by knowing the seed state and the algorithm used. These methods are quicker than getting "true" *randomness* from the environment.

The many applications of randomness have led to many different methods for generating random data. These methods may vary as to how unpredictable or statistically random they are, and how quickly they can generate random numbers. [Randomness]

1.3 Pseudorandomness

A pseudorandom process is a process that appears random but is not. Pseudorandom sequences typically exhibit statistical randomness while being generated by an entirely deterministic[1] causal process. Such a process is easier to produce than a genuine random one, and has the benefit that it can be used again and again to produce exactly the same numbers, useful for testing and fixing software.

To date there is no known method to produce true randomness, because due to the very nature of randomness, any factor determining the outcome would mean that it is not random at all. The random number generation functions provided in all software packages are therefore pseudorandom.

A pseudo-random variable is a variable which is created by a deterministic procedure which (generally) takes random bits as input. The pseudo-random string will typically be longer than the original random string, but less random (less entropic, in the information theory sense). This can be useful for randomized algorithms. [Pseudo]

[1] Deterministic here means that given the same initial seed, the generator will always produce the same output sequence

1.4 Pseudo Random Bit Generator

1.4.1 Definition (informal definition)

A pseudorandom bit generator (PRBG) is a deterministic algorithm which, given a truly random binary sequence of length *n*, outputs a binary sequence of length $l(n) > n$ which *appears* to be random, with $l()$ being a polynomial. The input to the PRBG is called the seed, while the output is called a pseudorandom bit sequence. [Junod]

The output of a PRBG is not random; in fact the number of possible output sequences is at most a small fraction, namely $2^k/2^l$, of all possible binary sequences of length *l*. The intent is to take a small truly random sequence of much larger length, in such a way that an adversary cannot efficiently distinguish between output sequences of the PRBG and truly random sequences of length $l(n)$.

1.4.2 Definition

A pseudorandom bit generator is said to pass all polynomial-time[2] statistical tests if no polynomial-time algorithm can correctly distinguish between an output sequence of the generator and a truly random sequence of the same length with probability significantly greater than $\frac{1}{2}$

1.4.3 Definition

A pseudorandom bit generator is said to pass the next-bit test if there is no polynomial-time algorithm which, on input of the first $l(n)$ bits of an output sequence s, can predict the $(l+1)^{st}$ bit of *s* with probability significantly greater than $\frac{1}{2}$.

Definition 1.4.2 and 1.4.3 are equivalent.

1.4.4 Definition

A PRBG that passes the next-bit test (possibly under some plausible but unproved mathematical assumption such as the intractability of factoring integers) is called a cryptographically secure pseudorandom bit generator (CSPRBG)

2 Pseudorandom bit generation

A one-way function *f* can be utilized to generate pseudorandom bit sequences (see Appendix A) by first selecting a random seed s, and then applying the function to the sequence of values $s, s+1, s+2, \ldots$; the output sequence is $f(s), f(s+1), f(s+2), \ldots$.

[2] The running time of the test is bounded by a polynomial in the length $l(n)$ of the outputsequence

Depending on the properties of the one-way function used, it may be necessary to only keep a few bits of the output values *f(s+i)* in order to remove possible correlations between successive values. Examples of suitable one-way functions f include a cryptographic hash function such as SHA-1, or a block cipher such as DES with secret key k. Although such ad-hoc methods have not been proven to be cryptologically secure.

In order to make sure that such generators are secure, they should be subject to a variety of statistical tests designed to detect the specific characteristics expected of random sequences. Passing these statistical tests is a necessary but not sufficient condition for a generator to be secure.

Derrick Henry Lehmer invented the linear congruential generator, used in most pseudorandom number generators today in 1951. [Pseudo]

The linear congruential generator is an example of an unsecure PRGB, which produces a pseudorandom sequence of numbers $x_1, x_2, x_3, \ldots$ according to the linear reccurence

$$x_n = ax_{n-1} + b \bmod m, \quad n \geq 1;$$

integers *a,b*, and *m* are parameters which characterize the generator, while x_0 is the (secret) seed. While such generators are commonly used for simulation purposes and probabilistic algorithms, and pass the statistical tests, they are predictable and hence entirely insecure for cryptographic purposes: given a partial output sequence, the remainder of the sequence can be reconstructed even if the parameters *a, b*, and *m* are unknown. [Stinson]

3 Cryptographically secure random bit generation

For such applications as cryptography, the use of pseudorandom number generators (whether hardware or software or some combination (See Appendix B)) is insecure. When random values are required in cryptography, the goal is to make a message as hard to crack as possible, by eliminating or obscuring the parameters used to encrypt the message (the key) from the message itself or from the context in which it is carried. Pseudorandom sequences are deterministic and reproducible; all that is required to discover and reproduce a pseudorandom sequence is the algorithm used to generate it and the initial seed. So the entire sequence of numbers is only as powerful as the randomly chosen parts - sometimes the algorithm and the seed, but usually only the seed.

Users and designers of cryptography are strongly cautioned to treat their randomness needs with the utmost care. Absolutely nothing has changed with the era of computerized cryptography, except that patterns in pseudorandom data are easier to discover than ever before. Randomness is, if anything, more important than ever. [Pseudo]

The security of a CSPRBG relies on the presumed intractability of an underlying number-theoretic problem. The modular multiplications that these generators use make them relatively slow compared to the (ad-hoc) pseudorandom bit generators. [Menezes et. al]

Other known CSPRBG are the RSA generator and Micali-Schnorr describen in [Menezes et. al]

4 Next bit predictors

A useful concept in studying bit generators is that of a next bit predictor, which works as follows. Let *f* be a *(k,l)* -bit generator. Suppose $1 \leq i \leq l-1$, and we have a function **nbp** : $(\mathbb{Z}_2)^{i-1} \rightarrow \mathbb{Z}_2$, which takes as input an *(i-1)* -tuple $z^{i-1} = (z_1, \ldots, z_{i-1})$, which represents the first *i-1* bits produced by *f* (given an unknown, random, *k*-bit seed). Then the function **nbp** attempts to predict the next bit produced by *f*, namely z_i. We say that the function **nbp** is an ε-*i*th bit predictor if **nbp** can predict the *ith* bit of the generated bitstring (given the first *i-1* bits) with probability at least $\frac{1}{2} + \varepsilon$, where $\varepsilon > 0$.

Theorem 1:

Let I be a *(k,l)* -bit generator. Then the function **nbp** is an ε-*ith* bit predictor for *f* if and only if

$$\sum_{z^{i-1} \in (\mathbb{Z}_2)^{i-1}} \left(pf\left(z^{i-1}\right) \times \Pr\left[z_i = nbp\left(z^{i-1}\right) | z^{i-1}\right]\right) \geq \frac{1}{2} + \varepsilon$$

Proof:

The probability of correctly predicting the *i*th generated bit is computed by summing (over all possible *(i-1)*-tuples $z^{i-1} = (z_1, \ldots, z_{i-1})$) the product of the probability that the *(i-1)*-tuple z^{i-1} is produced by the bit generator *f* and the probability that the *i*th bit is predicted correctly, given the *(i-1)*-tuple z^{i-1}.

Any predicting algorithm will predict any bit of a truly random bitstring with probability $\frac{1}{2}$, and thus we use the expression ½ + ε above. If a generated bitstring is not truly random, then it may be possible to predict a given bit with higher probability.

A next bit predictor can be used to construct a distinguishing algorithm. Suppose that **nbp** is an ε-*i*th bit predictor for a given integer $i \leq l$. The input to the algorithm is a sequence of *i* bits, denoted by z^i. It uses the function **nbp** to predict z_i, given the first *i-1* bits in the sequence. If the predicted value is the same as the actual value of z_i, then the algorithm outputs "1", otherwise the output is "0". The algorithm can be made as a polynomial-time Turing reduction.

The theorem below [Stinson] states that the above mentioned algorithm is a good distinguisher provided that **nbp** is a good *i*th bit predictor.

Theorem 2: Suppose **nbp** is a (polynomial-time) ε-*i*th bit predictor for the *(k,l)*-bit generator *f*. Let p_f be the probability distribution induced on $(\mathbb{Z}_2)^i$. Then the algorithm is a (polynomial-time) ε-distinguisher of p_f and p_u.

We will not prove this statement.

A main result in the theory of pseudo-random bit generators, is that a next bit predictor is a *universal test.* That is, a bit generator is "secure" if and only if there does not exist any polynomial time ε-*i*th bit predictor for the generator, except for very small values of ε. To show that the existence of a distinguisher implies the existence of a certain *i*th bit predictor, we use theorem 3 [Stinson]

Theorem 3: Suppose **dst** is a (polynomial-time) ε-distinguisher of p_f and p_u, where p_f is the probability distribution induced on $(\mathbb{Z}_2)^l$ by the *(k,l)*-bit generator *f*, and p_u is the uniform probability distribution on $(\mathbb{Z}_2)^l$. then for some *i*, $\mathbf{1} \leq i \leq l - \mathbf{1}$, there exists a (polynomial-time) $\frac{\epsilon}{l}$-*i*th bit predictor for *f*.

Proof: For $0 \leq i \leq l$ define q_i to be the probability distribution on $(\mathbb{Z}_2)^l$ where the first *I* bits are generated using the bit generator *f* (assuming that the *k*-bit seed is chosen uniformly at random), and the remaining *l-1* bits are generated uniformly and independently at random. Thus $q_0 = p_u$ and $q_l = p_f$. We are given that

$$|E_{dst}(q_0) - E_{dst}(q_l)| \geq \epsilon$$

By applying the triangle inequality, we have that

$$\epsilon \leq |E_{dst}(q_0) - E_{dst}(q_l)| \leq \sum_{i=1}^{l} |E_{dst}(q_{i-1}) - E_{dst}(q_i)|$$

Hence it follows that there is at least one value *i*, $1 \leq i \leq l$, such that

$$|E_{dst}(q_{i-1}) - E_{dst}(q_i)| \geq \frac{\epsilon}{l}$$

We will assume that

$$E_{dst}(q_{i-1}) - E_{dst}(q_i) \geq \frac{\epsilon}{l}$$

Which is a proof for theorem 5 in the section about Blum Blum Shub.

5 Blum Blum Shub – a pseudorandom number generator

The Blum Blum Shub (BBS) generator is almost impossible to avoid when talking about pseudorandom generators. It is often referred to because of its provable cryptographic security.

It works by generating long sequences of bits from a small core sequence – the seed. The generator is based on the next-bit-predictor idea which means that it is almost impossible to guess the next bit from the preceding one. This will be discussed more thoroughly in the forthcoming section on proving the security of the generator.

5.1 The algorithm

To understand what is going on in the algorithm, we start by making a definition of quadratic residues and quadratic non-residues.

5.1.1 Definition (Quadratic residues)

Let n be an odd prime. Then a number $q \in \mathbb{Z}_n^*$ is a quadratic residue if there exists an x$\in$ $\mathbb{Z}_n$ such that

$$q \equiv x^2 \pmod{n}.$$

If no such x exists q is a quadratic non-residue.

The set of quadratic residues mod n is denoted QR(n), and the set of quadratic non-residues mod n be denoted QNR(n).

This is used in the algorithm, and will be discussed more in the proof of security of the algorithm.

The algorithm (figure 1) is quite simple and it works as follows:

Let n = q · p where q and p are both primes and $p \equiv q \equiv 3 \pmod{4}$, and let QR(n) be the set of quadratic residues modulo n. The seed s_0 to generate the whole sequence from is taken from this QR(n) – in other words $s_0 \in$ QR(n). Note that to make sure s_0 belongs to QR(n) we can choose an element $s_{-1} \in \mathbb{Z}_n^*$ and then compute $s_0 = s_{-1}^2 \bmod n$.

First we generate a sequence of numbers from the seed $s_1, s_2, ..., s_l$. To generate the next number in the sequence we use the previous number and compute the square root of this mod n, so we have that $s_{i+1} = s_i^2 \bmod n$ where $0 \le i \le l-1$ and l is the length of the generated sequence of numbers.

When having achieved this sequence we need to make it into bits. This is done by taking modulo n on each s_i, so we have that $f(s_0) = (z_1, z_2, ..., z_l)$ where $z_i = s_i \pmod{n}$ and $1 \le i \le l$.

Putting all these things together we get that $z_i = (s_0^{2^i} \bmod 2), 1 \le i \le l$.

Blum Blum Shub Algorithm

Generate the two primes p, q such that $p \equiv q \equiv 3 \bmod 4$.

n := p·q

Generate the seed s_0 randomly so that s∈ QR(n).

$x := s^2 \bmod n$

for i = 0,…,l **do**

    $x := x^2 \bmod n$

    z := x mod 2

    output z

end for

Figure 1

5.2 Security

Before we prove the security of BBS we will look at some number-theoretic facts that we have to use in the proof. The proof is based mostly on [Stinson], but [junod] and [Greve et al.] is used as clarification.

5.2.1 Definition (Legendre symbol)

Let p be an odd prime. For any integer a∈ $\mathbb{Z}_p^*$ we define the Legendre symbol as follows:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } a \equiv 0 \pmod{p} \\ 1 & \text{if } a \text{ is quadratic residue} \bmod p \\ -1 & \text{if } a \text{ is a quadratic non-residue} \bmod p \end{cases}$$

To compute the Legendre symbol of an element a∈ $\mathbb{Z}_p$, we can use the following theorem:

Theorem 4 (Computing Legendre symbols): Let p be an odd prime, and let a∈ $\mathbb{Z}_p^*$. Then

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}$$

Proof: Let a ∈ QR(p) so that $a = b^2 \pmod p$, where b∈ $\mathbb{Z}_p^*$. Then we have that

$$a^{\frac{p-1}{2}} \equiv (b^2)^{\frac{p-1}{2}} \equiv b^{p-1} \equiv 1 \pmod{p}$$

This is because of Fermat's Little Theorem that says: $a^{p-1} \equiv 1 \pmod{p}$.

Now, let a ∈ QNR(p). Then we let g be generator of the cyclic group $\mathbb{Z}_p^*$ of order p-1.

For t = 2s + 1 where t is odd, we let $a = g^t$. Otherwise $a = g^t = g^{2s} = (g^s)^2$. We also have that

$$a^{\frac{p-1}{2}} \equiv g(t)^{\frac{p-1}{2}} \equiv (g^{2s})^{\frac{p-1}{2}} \cdot g^{\frac{p-1}{2}} \equiv g^{\frac{p-1}{2}} \pmod{p}$$

Now, since $(g^{\frac{p-1}{2}})^2 = 1$ we have that $g^{\frac{p-1}{2}} \in \{-1,1\}$.

While g is a generator of $\mathbb{Z}_p^*$, the order of g is p-1 and $g^{\frac{p-1}{2}} = -1$.

□

More generalised than the Legendre symbol we have the Jacobi symbol.

5.2.2 Definition (Jacobi symbol)

Let n be an odd positive integer, and the prime power factorization of n be: $n = \prod_{i=1}^{k} p_i^{e_i}$.

Let a be an integer. Then the Jacobi symbol is

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{e_i}$$

If p and q are two odd primes and p ≠ q, and we let n = p·q, then we have from definition 5.2.2 that

$$\left(\frac{x}{n}\right) = \begin{cases} 0 & \text{if } \gcd(x,n) > 1 \\ 1 & \text{if } \left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = 1 \text{ or if } \left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = -1 \\ -1 & \text{if one of } \left(\frac{x}{p}\right) \text{ and } \left(\frac{x}{q}\right) = 1 \text{ and the other} = -1 \end{cases}$$

We know that x is a quadratic residue if and only if $\left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = 1$.

This gives us the next definition.

5.2.3 Definition (pseudo-square)

An element x ∈ $\widetilde{QR(n)}$ is a pseudo-square modulo n where $\widetilde{QR(n)}$ is defined as

$$\widetilde{QR(n)} = \left\{ x \in \mathbb{Z}_n^* \setminus QR(n) : \left(\frac{x}{n}\right) = 1 \right\}$$

and

$$\widetilde{QR(n)} = \left\{ x \in \mathbb{Z}_n^* : \left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = -1 \right\}$$

From this we can se that $|QR(n)| = \widetilde{QR(n)} = \frac{(p-1)(q-1)}{4}$.

One main component in the proof of security for BBS is the challenge of solving the problem called Composite Quadratic Residues problem.

Problem 1 (Composite Quadratic Residues): Let n = p· q where n is a positive integer, p and q are unknown odd primes and p ≠ q, and x ∈ $\mathbb{Z}_n^*$ so that $\left(\frac{x}{n}\right) = 1$. **Is x ∈ QR(n)?**

What the problem asks of us is to distinguish quadratic residues modulo n from pseudo-squares modulo n, but to do that is no more difficult than to factoring n. If we assume that we can find out p and q, then we know the factorization of n since n = p · q, and then we can easily compute $\left(\frac{x}{p}\right)$ (or $\left(\frac{x}{q}\right)$). Because $\left(\frac{x}{n}\right) = 1$ we also know that x ∈ QR(n) if and only if $\left(\frac{x}{p}\right)$ = 1.

The only way to solve the problem in a somewhat reasonable amount of time and efficiency is to know the factorization of n, and this makes our proof of BBS very strong. In BBS we know that n = p · q where $p \equiv q \equiv 3 \pmod 4$, and therefore for any x ∈ QR(n) there exists a unique $\sqrt{x}$ that also belongs to QR(n) (the principal square root). The mapping $x \mapsto x^2 \mod n$, which we can see is the same as used in BBS, is as a result of the above a permutation on QR(n).

To go more into detail on the security of BBS we will introduce a concept called previous bit predictor. It is almost the same as next bit predictor, which we have already looked at, but this time we will try to see if we can determine the seed instead of starting with the seed and try to predict the other bits. For the rest of the proof we will assume that n = p·q where we have that p ≠ q and both p and q are unknown primes so that we do not know the factorization of n. Furthermore $p \equiv q \equiv 3 \pmod 4$. About the bits generated by the BBS we will suppose they are ε-distinguishable from l random bits.

We will not prove the next theorem about previous bit predictor, since it is similar to the theorem about next bit predictors.

Theorem 5 (previous bit predictor): Suppose an ε-distinguisher (polynomial time) of p_f and p_u exists. p_f is here presumed to be the probability distribution on $(\mathbb{Z}_2)^l$ lead to by the (k, l)-BBS, f. p_u

is the uniform probability distribution on $(\mathbb{Z}_2)^l$. Then a polynomial time $\frac{\varepsilon}{l}$-previous bit predictor for f exists.

The next algorithm we will look at (figure 2) uses a δ-previous bit predictor pbp. It is a probabilistic algorithm, and it is supposed to distinguish quadratic residues modulo n from pseudo squares modulo n with probability $\frac{1}{2}+\delta$.

What it does is that it takes an input x – an element with Jacobi symbol equal to 1, that is $x \in QR(n) \cup \widetilde{QR(n)}$. It uses the BBS with x^2 as seed. Then we test if x ∈ QR(n). If x ∈ QR(n) it follows that x itself could have been used as seed, and the sequence we get as result from the BBS with x as seed would be the same as the one with x^2. If x instead belongs to $\widetilde{QR(n)}$ it would have been the same as using –x as input. The algorithm tests if x ∈ QR(n) as described in theorem 6 below, and it uses a pbp as oracle.

QR-Test(x, n) algorithm

$s_1 := x^2 \bmod n$ // where $s_1 \in$ QR(n)
$z_1 := s_1 \bmod 2$
Use BBS to compute $z_2, \ldots, z_l$ with s_1 as seed
external pbp
$z := pbp(z_1,\ldots,z_l)$
if((x mod 2) = z)
    then return "yes"
    else return "no"

Figure 2

Theorem 6 (QR-Test): Let pbp be a polynomial time δ-previous bit predictor for f, where f is a (k, l)-BBS generator. The algorithm QR-Test then returns the correct answer for quadratic residuosity

with probability at least $\frac{1}{2}+\delta$. This is in polynomial time. The probability is the average of any $x \in QR(n) \cup \widetilde{QR(n)}$ where x is uniformly distributed and chosen randomly.

Proof: It follows from the fact that n = p· q and $p \equiv q \equiv 3 \pmod 4$ that $\left(\frac{-1}{n}\right)=1$ which implies that -1 ∈ $\widetilde{QR(n)}$. Then if we let s_0 be the principal square root of s_1, and we suppose that $\left(\frac{x}{n}\right)=1$, we have that if x ∈ QR(n) then s_0 = x, but if x ∈ $\widetilde{QR(n)}$ instead s_0 = -x.

The sequence of bits $z_1,\ldots z_l$ of length l is the same as if we used BBS with input seed s_0 to generate it. Because n is odd we know that (-x mod n) mod 2 ≠ (x mod n) mod 2, and x ∈ QR(n) if and only if s_0 = x. This means that the QR-Test only gives the right answer if pbp can predict z, which gives us what we were looking for.

□

The QR-Test algorithm can be improved. It promised to give the correct answer with a probability as an average of any x, but in the algorithm we will describe now (figure 3) we promise to give the correct answer for all inputs with the same probability as the QR-Test, namely at least $\frac{1}{2}+\delta$. The algorithm is an unbiased Monte Carlo algorithm, which means that it may output an incorrect answer.

The MC-QR-Test(x) algorithm (as we call it) is basically a randomization procedure. It takes x and randomize it to a residue x'. The procedure of randomizing x is as follows: first it takes x and multiply it with a random quadratic residue, second it multiply the outcome of this with 1 or -1 randomly. Finally it uses the QR-Test algorithm to test if the result it comes up with is a quadratic residue.

MC-QR-Test(x) algorithm

x' := r^2· x mod n // where r ∈ $\mathbb{Z}_n^*$ is chosen randomly

x' := s· x' mod n // where s ∈ {-1, 1} is chosen randomly

external QR-Test

t := QR-Test(x')

if(((t = yes) && (s = 1)) || ((t = no) && (s = -1)))

then return "yes"

else return "no"

Figure 3

As with the QR-Test we have a theorem for MC-QR-Test:

Theorem 7 (MC-QR-Test): If we assume that the QR-Test with probability at least $\frac{1}{2}+\delta$ and in polynomial time can determine quadratic residuosity correctly, then the MC-QR-Test algorithm is a polynomial time Monte Carlo algorithm for the Composite Quadratic Residues problem with error probability at most $\frac{1}{2}-\delta$.

Proof: As input we give $x \in QR(n) \cup \widetilde{QR(n)}$. We would like to know if x is a quadratic residue. The algorithm makes an element x' which is a random element of $QR(n) \cup \widetilde{QR(n)}$. Since we know that we always make an element x' we can find out the status of x by ascertaining the status of x'.

□

The next theorem (theorem 8) shows a generalisation on unbiased Monte Carlo algorithms. It tells us that if the Monte Carlo algorithm has error probability at most $\frac{1}{2}-\delta$ (as MC-QR-Test has), we can make another algorithm with error probability at most γ, where $\gamma > 0$. This would give us a probability very close 1. If we run the Monte Carlo algorithm 2m + 1 times, where m is some integer, we can take the most frequent result as the final result. The theorem then tells us about the dependency between m and γ.

Theorem 8: If we have an unbiased Monte Carlo algorithm A with error probability at most $\frac{1}{2}-\delta$, and we define an algorithm A^n to be A run n times, where n = 2m + 1, and with output as the most frequent result of A, we have an error probability for A^n at most $\frac{(1-4\delta^2)^m}{2}$.

Proof: The probability of getting the correct answer exactly i times, when running A n times is at most:

$$\binom{n}{i}\left(\frac{1}{2}+\delta\right)^i\left(\frac{1}{2}-\delta\right)^{n-i}$$

The probability that outcome of A^n is incorrect (that is – that the most frequent answer is incorrect) is equal to the probability that i (the number of correct answers in the n runs of A) is at most m. We can then compute the probability:

$$
\begin{aligned}
\Pr[error] &\le \sum_{i=0}^{m} \binom{n}{i} \left(\frac{1}{2}-\delta\right)^{i} \left(\frac{1}{2}-\delta\right)^{2m+1-i} \\
&= \left(\frac{1}{2}+\delta\right)^{m} \left(\frac{1}{2}-\delta\right)^{m+1} \sum_{i=0}^{m} \binom{n}{i} \left(\frac{\frac{1}{2}-\delta}{\frac{1}{2}+\delta}\right)^{m-i} \\
&\le \left(\frac{1}{2}+\delta\right)^{m} \left(\frac{1}{2}-\delta\right)^{m+1} \sum_{i=0}^{m} \binom{n}{i} \\
&= \left(\frac{1}{2}+\delta\right)^{m} \left(\frac{1}{2}-\delta\right)^{m+1} 2^{2m} \\
&= \left(\frac{1}{4}-\delta^{2}\right)^{m} \left(\frac{1}{2}-\delta\right) 2^{2m} \\
&= \left(1-4\delta^{2}\right)^{m} \left(\frac{1}{2}-\delta\right) \\
&\le \frac{\left(1-4\delta^{2}\right)^{m}}{2}
\end{aligned}
$$

This was what we wanted to show.

□

We can lower the error probability to a value γ where $0 < \gamma < \frac{1}{2} - \delta$.

When we choose our m we have to make sure that $\frac{(1-4\delta^2)^m}{2} \le \gamma$ which gives us

$$m = \left\lceil \frac{1+\log_2 \gamma}{\log_2(1-4\delta^2)} \right\rceil$$

If A is run 2m + 1 times, the output will be the correct answer with probability at least 1 - γ. Since m is at most $\frac{c}{(\gamma\delta^2)}$, where c is a constant. That means that the number of times we have to run A is polynomial in $\frac{1}{\gamma}$ and $\frac{1}{\delta}$.

We have now looked at an algorithm BBS and some ways to reduce it. Figure 4 shows the reductions. It is taken from [Stinson] page 344. The reductions we have done are all polynomial time algorithms, and since it is commonly accepted that there are no polynomial time algorithms to

solve the Composite Quadratic Residues problem without compromising on error probability, we have proven the security of the BBS generator as intended.

Reductions

(k, l)-BBS Generator can be ε-distinguished from l random bits

(ε/l)-previous bit predictor for (k, l)-BBS Generator

distinguishing algorithm for Composite Quadratic Residues
that is correct with probability at least ½ + ε/l

unbiased Monte Carlo algorithm for Composite Quadratic Residues
having error probability at most ½ - ε/l

unbiased Monte Carlo algorithm for Composite Quadratic Residues
having error probability at most γ, for any $\gamma > 0$.

It is possible to improve the BBS generator compared to the described above. If we instead of taking the least significant bit of each s_i when we compute $s_i = s_0^{2^i} \bmod n$, take the r least significant bits from each s_i, where r is a positive integer, we will improve the efficiency by a factor r. Of course we need to make sure that the algorithm is still secure, but it has been shown that as long as $r \leq \log_2 \log_2 n$ there will be no security problem.

6 Conclusion

In this report we have discussed the importance of being able to produce pseudorandom bits when talking about cryptology. We have seen how to generate these bits using pseudorandom bit generators, and we have particularly looked in detail at the Blum Blum Shub generator, which we have proven to be cryptographically secure. In our discussion of BBS, we have seen more general areas in the generation of random bits, like the next bit theorem. Finally we have shortly seen an improvement to BBS.

Bibliography

[Menezes et. al] Handbook of applied Cryptography, by *A. Menezes, P. van Oorschot band S. Vanstone*, CRC Press, 1996

[Stinson]Cryptography, theory and practice 3rd ed., by *Douglas R. Stinson*, Chapman & Hall/CRC, 2006

[Randomness] http://en.wikipedia.org/wiki/Randomness, retrieved December 18th 2007

[Pseudo] http://en.wikipedia.org/wiki/Pseudorandomness, retrieved December 18th , 2007

[BS] http://en.wikipedia.org/wiki/Pseudo-random_binary_sequence, retrieved December 18th, 2007

[Junod] http://crypto.junod.info/bbs.pdf, retrieved December 18th, 2007

Appendices

A Pseudo Random Bit Sequence

A PRBS is random in a sense that the value of an a_j element is independent of the values of any of the other elements, similar to real random sequences.

A binary sequence (BS) is a sequence of N bits,

a_j for $j = 0,1,...,N-1$,

i.e. m ones and $N-m$ zeros. A BS is pseudo-random (PRBS) if its autocorrelation function

$$C(v) = \sum_{j=0}^{N-1} a_j a_{j+v}$$

has only two values:

$$C(v) = \begin{cases} m, & \text{if } v \equiv 0 \pmod{N} \\ mc, & \text{otherwise} \end{cases}$$

where

$$c = \frac{m-1}{N-1}$$

is called the *duty cycle* of the PRBS.

Because it is deterministic it is only pseudo random. After N elements it starts to repeat itself, unlike real random sequences, such as sequences generated by radioactive decay or by white noise. The PRBS is more general than the n-sequence, which is a special pseudo-random binary sequence of n bits generated as the output of a linear shift register. An n-sequence always has a 1/2 duty cycle and its number of elements $N = 2^k - 1$. PRBS's are used in telecommunication, encryption, simulation, correlation technique and time-of-flight spectroscopy. [BS]

B Random bit generation

A true random bit generator requires a naturally occurring source of randomness. Designing a hardware device or software program to exploit this randomness and produce a bit sequence that is free of biases and correlations is a difficult task. Additionally, for most cryptographic applications, the generator must not be subject to observation or manipulation by an adversary.

B.1.1 Hardware-based generators

Hardware-based random bit generators exploit the randomness which occurs in some physical phenomena. Such physical processes may produce bits that are biased or correlated, in which case they should be subjected to de-skewing techniques mentioned in **Fejl! Henvisningskilde ikke fundet.**. Examples of physical phenomena include:

1. Elapsed time between emission of particles during radioactive decay;
2. Air turbulence within a sealed disk drive which causes random fluctuations in disk drive sector read latency times.

B.1.2 Software-based generators

Designing a random bit generator in software is even more difficult than doing so in hardware. Processes upon which software random bit generators may be based include:

1. The system clock;
2. Elapsed time between keystrokes or mouse movement;
3. Operating system values such as system load and network statistics.

The behavior of such processes can vary considerably depending on various factors, such as the computer platform. It may also be difficult to prevent an adversary from observing or manipulating these processes. A well-designed software random bit generator should utilize as many good sources of randomness as are available. Using many sources guard against the possibility of a few sources failing, or being observed or manipulated by an adversary. Each source should be sampled, and the sampled sequences should be combined using a complex mixing function; one recommended technique for accomplishing this is to apply a cryptographic hash function such as SHA-1 or MD5 to a concatenation of the sampled sequences. The purpose of this is to distill the (true) random bits from the sampled sequences.

B.1.3 De-skewing

A natural source of random bits may be defective in that the output bits may be biased (the probability of the source emitting a 1 is not equal to $\frac{1}{2}$) or correlated (the probability of the source emitting a 1 depends on the previous bits emitted). There are various techniques for generating truly random bit sequences from the output bits of such a defective generator; such techniques are called de-skewing techniques. [Menezes et. al]

Katrine Mollerup Johansen, 20042707
Lene Mejlby, 20051795