A Cryptographically Secure Random Number Generator for Maple

Michael Monagan and Greg Fee*
Centre for Experimental and Constructive Mathematics
Simon Fraser University,
Burnaby, B.C., V5A 1S6, Canada.
mmonagan@cecm.sfu.ca, gfee@cecm.sfu.ca

Abstract

The best known pseudo-random number generators are the linear congruential generator (LCG) and the linear feedback shift register (LFSR). Neither, however, is good for cryptographic applications. The Blum-Blum-Shub (BBS) generator is one of the first and best known cryptographically secure pseudo-random bit generators. If p and q primes and $n = pq$, the security of the BBS generator assumes that deciding quadratic residuosity in $\mathbb{Z}_n$ and factoring n are computationally infeasible.

In this paper we show why an LCG is not good for cryptographic applications. We sketch the proof of the security of the BBS generator and show how to choose the primes so that the period of the BBS generators is guaranteed to be long.

We then construct BBS generators for Maple using primes of length 512, 768 and 1024 bits with guaranteed long period. We also support primes of length 32, 48 and 64 bits. While not cryptographically secure, these BBS generators are expected to yield pseudo-random bits of high quality and fast enough to be competitive with Maple's built-in LCG.

*The work was supported by the MITACS NCE of Canada.

1 Introduction

Definition 1.1 A *pseudo random number generator*, PRNG for short, is a function $F : \mathbb{Z}_m \to \mathbb{Z}_n^L$ where m is strictly less than n^L. The input $x_0 \in \mathbb{Z}_m$ is called the *seed* of the PRNG and the output sequence of L numbers $x_1, x_2, ..., x_L$ is called the *pseudo-random number sequence*. If $n = 2$ we will call F a *pseudo-random bit generator* or PRBG for short.

Example 1.2: Perhaps the PRNG most often used is the linear congruential generator, or LCG for short. An LCG is based on modular arithmetic. Let n be a positive integer, $a, b \in \mathbb{Z}_n$. For $k > 0$ define

$$x_k = ax_{k-1} + b \bmod n.$$

Thus the next number in the pseudo-random number sequence is a linear function of the previous one. If n is chosen to be a prime, a is chosen to be a primitive element in $\mathbb{Z}_n$ and b is set to 0, and $x_0 \neq 0$, then the pseudo-random-number sequence generated by the LCG has period $\pi = n - 1$. Maple's pseudo-random number generator `rand` is an LCG of this form.

Example 1.3: An example of a PRBG which can be very efficiently implemented in hardware is the linear feedback shift register, or LFSR for short. It is based on the following linear recurrence. Let m be a positive integer and let $c_0, c_1, ..., c_{m-1} \in \mathbb{Z}_2$ with $c_0 = 1$. For $k \geq 0$ define

$$x_{k+m} = c_0 x_k + c_1 x_{k+1} + ... + c_{m-1} x_{k+m-1} \bmod 2.$$

Then if the seed $x_0, x_1, ..., x_{k-1} \in \mathbb{Z}_2$ are not all 0, and $m(y) = c_0 + c_1 y + ... + c_{m-1} y^{m-1} + y^m$ is a primitive polynomial in $\mathbb{Z}_2[y]$, that is, $m(y)$ is irreducible and y is a primitive element in the finite field $\mathbb{Z}_2[y]/(m)$, then the period of the pseudo-random bit sequence is $2^m - 1$.

Since $m < n^L$, the output of a PRNG cannot be truly random because not all possible pseudo-random number sequences are possible. What one aims to do in constructing a good PRNG is that the pseudo-random number sequence "appears to be random" in the sense that it will pass known statistical tests for randomness. For example, in a long pseudo-random bit sequence $x_1, x_2, ..., x_L$, the bit sequence 0101 should appear with a certain probability distribution. A necessary but not sufficient requirement for a PRNG to be good is that the period π of the PRNG be sufficiently large.

It is rather surprising to find that LCGs and LFSRs produce pseudo-random number sequences that are very good for many applications. How good are the pseudo random numbers generated by Maple's LCG `rand` ? Knuth [1] gives a table comparing, using the spectral test, LCGs with several values of n, e.g. $2^{32}, 2^{35}, 10^{10}$. Based on the spectral test, Maple's PRNG rand is significantly better than all the LCGs in Knuths table except one with $n = 2^{64}$ where it is comparable.

However, LCGs are not secure for cryptographic purposes. The reason is that if an adversary can obtain a consecutive sequence of sufficiently many x_i's he will be able, in polynomial time, to determine n, a and b, and hence all of the x_i's. This may enable him to break a cryptographic protocol. In this sense, an LCG is not random at all. For if a sequence is truly random, one should not be able, given $x_1, x_2, ..., x_L$, to predict x_{L+1} with probability significantly greater than $1/n$. LFSRs are not secure either for cryptographic purposes. We will show how one can easily attack an LCG using Maple's LCG as an example. Notice that if $b = 0$ then

$$x_i = ax_{i-1} \bmod n = a^2 x_{i-2} \bmod n = x_0 a^i \bmod n.$$

Now suppose we obtain x_1, x_2, x_3, x_4, x_5. Hence

$$x_1 x_4 \equiv x_0^2 a^5 \equiv x_2 x_3 \bmod n$$

hence $n | x_1 x_4 - x_2 x_3$. Also $n | x_1 x_3 - x_2^2$. Thus

$$g = \gcd(x_1 x_4 - x_2 x_3, x_1 x_3 - x_2^2)$$

will be a multiple of n. Since n is a prime and $0 < x_i x_j < n^2$ then n is the largest prime dividing g, hence n can be obtained by factoring g. However, we do not need to factor g which might be difficult if n is large. Instead we take the gcd with more differences, so that g is likely to be a small multiple of n. Once we have found n we can determine a from $a = x_2 x_1^{-1} \bmod n$. And once we have n and a we have $x_0 = x_1 a^{-1} \bmod n$ the seed. Trying this in Maple we find

```
> for i to 5 do x[i] := rand() od;
```

```
                    x[1] := 427419669081
                    x[2] := 321110693270
                    x[3] := 343633073697
```

```
                                  x[4] := 474256143563
                                  x[5] := 558458718976

> g := igcd( x[1]*x[4]-x[2]*x[3], x[1]*x[3]-x[2]^2 );
                                    g := 999999999989

> n := igcd( g, x[1]*x[5]-x[2]*x[4] );
                                  n := 999999999989
> a := x[2]/x[1] mod n;
                                  a := 427419669081
> x[0] := x[1]/a mod n;
                                      x[0] := 1
```

Thus the modulus used by Maple's LCG is $n = 999,999,999,989 = 10^{12} - 11$, the multiplier $a = 427,419,669,081$, and the seed $x_0 = 1$.

1.1 The Blum-Blum-Shub Generator

The following definition for what it means for a PRNG to be pseudo-random seems to be the right one.

Definition 1.1 A pseudo-random-bit generator is said to be *cryptographically secure* if given a consecutive sequence of l output bits $x_i, x_{i+1}, ..., x_{i+l-1}$ of the PRBG, it is computationally infeasible to determine x_{i-1} or x_{i+l} with probability significantly greater than $1/2$.

Here *computationally infeasible* is understood to mean not polynomial time in certain parameters and *significantly greater than 1/2* may, for our purposes here, may be replaced by $1/2 + \epsilon$ where $0 < \epsilon \leq 1/2$ is a constant.

The cryptographically secure PRBG that we look at in this paper is the Blum-Blum-Shub generator, or BBS generator for short. See [3] for the original paper. We refer the reader to Chapter 12 of Stinson's book [2] for an accessible reference. Let n be a product of two large primes p and q both congruent to 3 modulo 4. The Blum-Blum-Shub generator works as follows;

Algorithm BBS
Input x_0 a quadratic residue in $\mathbb{Z}_n$
Set $x_i = x_{i-1}^2 \bmod n$ for $1 \leq i \leq L$.
Set $z_i = x_i \bmod 2$ for $1 \leq i \leq L$.
Output $z_1, z_2, ..., z_L$.

See section 2 for a definition of the quadratic residues. The cryptographic security of the BBS generator is based on the difficulty of testing whether $x \in \mathbb{Z}_n$ is a quadratic residue (a square) modulo n. Note this is no more difficult than factoring n for if the factorization of n is known, the quadratic residuosity of x in $\mathbb{Z}_n$ is easily decidable. Thus the primes p and q need to be large enough such that it is computationally infeasible to factor n. As of the year 2003 that means that p and q should be 512 bit primes or, for longer term security, 768 or 1024 bit primes. In section 3 we will formalize these ideas more precisely and sketch the proof of security of the BBS generator. The basic idea though is quite simple; we will prove that given $z_2, z_3, ..., z_l$ it is computationally infeasible to determine z_1. More precisely, we will prove that if you could do this in polynomial time then we would have a polynomial time algorithm to decide if any $x \in \mathbb{Z}_n$ is a quadratic residue. It follows that the BBS generator should pass all statistical tests, including those not yet invented. For if a statistical test could predict z_1 with probability significantly greater then $1/2$, then essentially that test would yield a polynomial time algorithm for the quadratic residue problem. The security of the BBS generator is therefore not absolute but it does say that the breaking of the BBS generator is at least as difficult as a well studied problem which is thought to have no efficient solution.

To use a BBS generator one must first choose x_0, the seed in such a way that the generator has a sufficiently long period. Since the user may not know the factorization of n, and hence cannot determine the period, this needs to be done with care so that the user does not, unknowingly, choose a seed with a short period.

In section 4 we show that if one chooses the primes p and q for use in the BBS generator to be of the form $2r + 1$ where r is a prime and 2 is a primitive element in $\mathbb{Z}_r$, then provided $x_0 \neq 1$, we will get a pseudo-random number sequence with a long period. Finding 512, 768, and 1024 bit primes of this form is computationally difficult. We show how to do this using Maple in section 5 and we give Maple code for our implementation of the BBS generator. Here is an example to illustrate our routine using 512 bit primes. The Maple subroutine `BBS` (see appendix) outputs a Maple subroutine which when called generates a sequence of L random bits, and when called again, generates the next L random bits, and so on.

```
> B := BBS[512](10,4);
      B := proc() x := irem(x^2, n); T[irem(x, 1024)] end proc
```

```
> B(); # 10 bits
                    1, 0, 0, 0, 1, 1, 0, 1, 0, 1

> B(); # next 10 bits
                    1, 1, 0, 0, 0, 0, 0, 1, 0, 1
```

Our generator also allows for primes of length 32, 48 and 64 bits. Although not cryptographically secure, these generators should produce pseudo-random bits of high quality in comparison with LCGs and other PRBGs.

Our paper is organized as follows. Section 2 presents background material on quadratic residues for understanding the results in sections 3 and 4. Section 3 sketches the proof of the cryptographic security of the BBS generator. Section 4 shows how to choose the primes for the BBS generator so that the period will be long. Section 5 details our Maple implementation and compares the efficiency of our BBS generators in Maple with Maple's LCG.

2 The Quadratic Residues in $\mathbb{Z}_n$

Let $\mathbb{Z}_n^* = \{x \in \mathbb{Z}_n \mid \gcd(x,n) = 1\}$ denote the set of invertible integers modulo n. We recall the definition of the quadratic residues in $\mathbb{Z}_n$.

Definition 2.1: *Let n be a positive integer. The set of quadratic residues of $\mathbb{Z}_n$ is $QR(n) = \{x^2 \bmod n | x \in \mathbb{Z}_n^*\}$.*

Let p and q be odd primes and let $n = pq$. It is not difficult to see that the number of quadratic residues in $\mathbb{Z}_p$ is $|QR(p)| = (p-1)/2$. Applying the Chinese remainder theorem we have $|QR(n)| = (p-1)(q-1)/4$. For example, QR(5) = {1,4}, QR(7) = {1,2,4} and QR(35) = {1,4,9,11,16,29}. We now look at some tests for whether x is a quadratic residue.

Lemma 2.2 (Euler's Criterion): *Let p be an odd prime and let $x \in \mathbb{Z}_p^*$.* Then $x \in QR(p) \iff x^{(p-1)/2} \equiv 1 \bmod p$.

Since the power $x^{(p-1)/2} \bmod p$ can be computed in polynomial time, in $O(\log_2 p)$ multiplications in $\mathbb{Z}_p$ using the well known repeated squaring algorithm, it follows that we can decide quadratic residuosity in $\mathbb{Z}_p$ in polynomial time. Again, applying the Chinese remainder theorem we have $x \in QR(n) \iff x \in QR(p)$ and $x \in QR(q)$.

Definition 2.3 The Jacobi symbol of $x \in \mathbb{Z}_p^*$ is defined to be

$$J(x/p) = +1 \text{ if } x \in QR(p), \quad \text{and} \quad -1 \text{ otherwise.}$$

The Jacobi symbol is multiplicative, that is, it satisfies

$$J(x/pq) = J(x/p)J(x/q).$$

Therefore in $\mathbb{Z}_n^*$ we have $J(x/pq) = +1$ if either $J(x/p = +1$ and $J(x/q) = +1$ or $J(x/p = -1$ and $J(x/q) = -1$. The Jacobi symbol can be computed in polynomial time using the Euclidean algorithm and hence it is a much more efficient test for quadratic residuosity than the test based on Euler's criterion. This is the algorithm that Maple's `numtheory[jacobi]` routine uses.

Definition 2.4 We define the *pseudo-squares* by

$$\tilde{QR}(n) = \{x \in \mathbb{Z}_n^* | J(x/p) = -1 \textit{ and } J(x/q) = -1\}.$$

For example, $\tilde{QR}(35) = \{3, 12, 13, 17, 27, 33\}$. From the above we see that $J(x/n)$ does not distinguish quadratic residues from pseudo-squares in $\mathbb{Z}_n$. We gather some additional properties about the set of quadratic residues and pseudo squares in $\mathbb{Z}_n$ in the following lemma without proof.

Lemma 2.5: *Let p and q be odd primes congruent to 3 modulo 4 and $n = pq$. Let $w, x \in QR(n)$ and $y, z \in \tilde{QR}(n)$. Then $wx \in QR(n)$, $wy \in \tilde{QR}(n)$, $yz \in QR(n)$, $-x \in \tilde{QR}(n)$ and $-y \in QR(n)$. Moreover, since also $1 \in QR(n)$ then $QR(n)$ is a subgroup of $\mathbb{Z}_n^*$.*

Problem 2.6: The Quadratic Residue Problem in $\mathbb{Z}_n$.
Let p and q be odd primes and $n = pq$. The quadratic residue problem in $\mathbb{Z}_n$ is: given $x \in \mathbb{Z}_n^*$ with $J(x/n) = +1$, is $x \in QR(n)$?

In the above we have seen that the Jacobi symbol $J(x/n) = +1$ tells us that $x \in QR(n) \cup \tilde{QR}(n)$ so the Jacobi symbol does not distinguish quadratic residues from pseudo-squares. If the factorization of n is known then $x \in QR(n)$ if and only if $x \in QR(p)$ and $x \in QR(q)$, equivalently, $J(x/p) = +1 = J(x/q)$. The problem, however, is that there is no known way to decide if $x \in QR(n)$ in polynomial time if the factorization of n is not known. It is not known if solving the quadratic residue problem is as hard as factoring n.

We now look at the related problem of computing the square roots. If $p \equiv 3 \bmod 4$ and $x \in QR(p)$ then the two square roots of x are given by the formula $\pm x^{(p+1)/4}$. If $p \equiv 1 \bmod 4$ and $x \in QR(p)$ then there is no simple formula but there is a random polynomial time algorithm for computing the two square roots, which is implemented by the `numtheory[msqrt]` command in Maple. If $x \in QR(n)$ then there are four square roots which can be computed using the Chinese remainder theorem if the factorization of n is known. Again, if the factorization of n is not known there is no known polynomial time algorithm to compute $\sqrt{x} \bmod n$. However, in this case, a polynomial time algorithm for finding square roots in $\mathbb{Z}_n$ yields a random polynomial time algorithm for factoring n – see [2].

Recall that the BBS generator computes the sequence of squares, i.e. sequence $x_1 = x_0^2 \bmod n, x_2 = x_1^2 \bmod n, x_3 = x_2^2 \bmod n, ...,$ where x_0 is chosen from $QR(n)$. Since the square of a quadratic residue is also a quadratic residue, this sequence must enter a cycle in QR(n). Hence we are interested in the sizes of the cycles in $QR(n)$ under the map $x \to x^2 \bmod n$. The reason that we are interested in these cycles is because if it should happen that the seed of the BBS generator x_0 is on a cycle of small period, the sequence of pseudo-random bits will not be cryptographically secure even if the general BBS generator is shown to be secure.

There are two possibilities that could occur, namely cycles which are simple loops and cycles of the form $x_1, x_2, ..., x_{j-1}, x_j, x_{j+1}, ..., x_r, x_j, ...$ Notice that in the latter case the quadratic residue x_j has two square roots, x_{j-1} and x_r, both of which are quadratic residues, and this does not happen in a cycle which is a closed loop. The following example illustrates that both cases do occur.

Example 2.5: The following two figures show the cycles in QR(55) and QR(77). For $n = 77$ we have one cycle of period 1, namely 1, one of period 2, namely (23,67), and three of period 4, namely (4,16,25,9), (15,71,36,64), and (37,60,58,53).

Recall that in the BBS generator the primes p and q are chosen to be congruent to 3 modulo 4. The following lemma says that only closed loops can occur when this requirement on the primes is made.

Lemma 2.6: *If p and q are primes both congruent to 3 modulo 4, $n = pq$ and $x \in QR(n)$ then exactly one square root of x is a quadratic residue in $\mathbb{Z}_n$.*

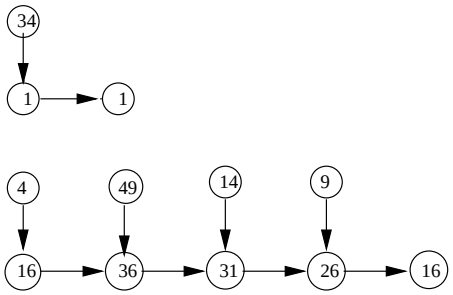

Figure 1: Cycles in QR(55)

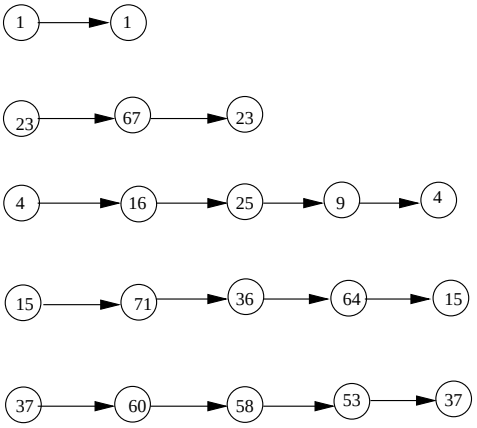

Figure 2: Cycles in QR(77)

Proof. We have $\pm z = \sqrt{x} = \pm x^{(p+1)/4} \bmod p$. Since x is a quadratic residue, $J(x/p) = 1$, and

$$J(+z/p) = J(+x^{(p+1)/4}/p) = J(x/p)^{(p+1)/4} = 1$$

and

$$J(-z/p) = J(-x^{(p+1)/4}/p) = J(-1/p)J(x/p)^{(p+1)/4} = (-1)^{(p-1)/2} = -1.$$

Hence $+x^{(p+1)/4} \bmod p$ is a quadratic residue modulo p and $-x^{(p+1)/4} \bmod p$ is a quadratic non-residue modulo p. Letting

$$\pm y = \sqrt{x} \bmod q = \pm x^{(q+1)/4} \bmod q$$

and applying the Chinese remainder theorem we have one square root (from $+y \bmod q$ and $+z \bmod p$) which is a quadratic residue in $\mathbb{Z}_n$.

3 The Security of the BBS Generator

Following Stinson [2] we sketch the proof that the BBS generator is cryptographically secure. Let p and q be two odd primes congruent to 3 modulo 4, $n = pq$ and $x \in QR(n) \cup \tilde{QR}(n)$. Let $BBS : \mathbb{Z}_n \to \mathbb{Z}_2^l$ be a Blum-Blum-Shub generator with input $x_0 \in QR(n)$ and output $z_1, z_2, ..., z_l$. The output bits come from $z_k = x_k \bmod 2$ for $1 \leq k \leq l$ and $x_{k+1} = x_k^2 \bmod n$ for $1 \leq k \leq l$. Consider the following problem; given n and $z_1, z_2, ..., z_l$, but not the factorization of n, determine $z_0 = x_0 \bmod 2$.

We first establish that this problem has a unique solution. Lemma 2.6 says that x_0 is the only square root of x_1 which is a quadratic residue. Secondly, if $y_k \in \tilde{QR}(n)$ such that $y_k^2 \equiv x_{k+1} \bmod n$ then $y_k \not\equiv x_k \bmod 2$, i.e. the least significant bit of the two square roots y_k and x_k of x_{k+1} differ. To see this first note that since $p \equiv q \equiv 3 \bmod 4$, it follows that

$$J(\frac{-1}{n}) = J(\frac{-1}{p})J(\frac{-1}{q}) = (-1)^{(p-1)/2}(-1)^{(q-1)/2} = (-1)^2$$

since $(-1)^{(p-1)/2} = (-1)^{(4k+2)/2} = -1$. Therefore

$$J(\frac{-x}{n}) = J(\frac{-1}{n})J(\frac{x}{n}) = J(\frac{x}{n}).$$

Hence if $x \in QR(n)$ then $-x \in \tilde{QR}(n)$. Since also n is odd implies $(-x \bmod n) = n - x \not\equiv x \,(\text{mod}2)$ the result follows.

Let O be a subroutine or oracle which on input of $z_1, z_2, ..., z_l$ can predict whether $z_0 = x_0 \bmod 2$ is 1 or 0 with probability greater than $1/2 + \epsilon$. Let $x \in QR(n) \cup \tilde{QR}(n)$. We will construct a Monte-Carlo algorithm which can decide whether $x \in QR(n)$ in probabilistic polynomial time in $\log n$ and $-\log \epsilon$. Remark: The reason the subroutine O is called an oracle is because we believe that it does not exist.

Algorithm A
Input: $x \in \mathbb{Z}_n^*$ *such that* $J(\frac{x}{n}) = 1$ *i.e.,* $x \in QR(n) \cup \tilde{QR}(n)$.
Output: $true \implies x \in QR(n)$, $false \implies x \in \tilde{QR}(n)$.
Compute $x_0 = x^2 \bmod n$ *and* $z_0 = x_0 \bmod 2$.

Compute $z_1, z_2, ..., z_l \leftarrow BBS(x_0)$
If $O(z_0, z_1, ..., z_{l-1}) \equiv x \bmod 2$ *then output true,*
otherwise output false.

It follows that if O predicts z_0 correctly with probability $1/2 + \epsilon$ then algorithm 2.3 outputs whether $x \in QR(n)$ with probability $1/2 + \epsilon$. The next idea is to randomize algorithm A so that we can make the probability of guessing whether $x \in QR(n)$ as high as we wish.

Algorithm B
Input: $x \in \mathbb{Z}_n^*$ *such that* $J(\frac{x}{n}) = 1$.
Output: $true \implies x \in QR(n)$, $false \implies x \in \tilde{QR}(n)$.
Choose $z \in \mathbb{Z}_n^*$ *at random.*
With probability 1/2 compute $y = r^2 x \bmod n$,
otherwise $y = -r^2 x \bmod n$.
If $A(y)$ *and* $y = r^2 x \bmod n$ *or not* $A(y)$ *and* $y = -r^2 \bmod n$
then output true, otherwise output false.

Algorithm B multiplies x by a random quadratic residue with probability $1/2$ and a random quadratic non-residue otherwise. Applying Lemma 3.3, if $x \in QR(n)$, then $r^2 x$ is a random quadratic residue and $-r^2 x \bmod n$ is a random pseudo-square. Similarly, if $x \in \tilde{QR}(n)$ then $r^2 x$ is a random pseudo-square and $-r^2 x$ is a random quadratic residue. It follows that if algorithm A determines quadratic residuosity correctly with probability at least $1/2 + \epsilon$ then algorithm B is a Monte-Carlo algorithm for the quadratic residue problem with error or failure probability at most $1/2 - \epsilon$. The last step is to run algorithm B many times, say $2m + 1$ times, and take the most frequent result to the the output. In other words, take the majority vote as the correct answer. One can show that if we do this then the error probability is at most

$$\frac{(1 - 4\epsilon^2)^m}{2}.$$

Hence, if we want the error probability to be less than some fixed constant, δ say, it suffices to take $m = \lceil \frac{1 + \log_2 \delta}{\log_2(1 - 4\epsilon^2)} \rceil$. For example, if $\epsilon = 1/4$ and $\delta = 10^{-10}$ then we have $m = 78$, hence, we need $2m + 1 = 157$ calls to algorithm B to reduce the error probability to below 10^{-10}.

In summary, if one could predict the *previous* bit of a consecutive sequence of l bits from a BBS generator with probability significantly greater than $1/2$, then we would have a probabilistic polynomial time algorithm for deciding quadratic residuosity in $\mathbb{Z}_n$. Since it is believed that no such algorithm exists, this provides some evidence that the bits generated by the Blum-Blum-Shub PRBG are cryptographically secure.

4 The Period of the BBS Generator

In this section we will show how to choose the primes p and q and then how the user, without knowledge of p and q, can choose the seed $x_0 \in QR(n)$ in such a way that the BBS sequence $x_0^2, x_0^4, x_0^8, ...$ will have a long period modulo n.

Let p and q be two odd primes both congruent to 3 modulo 4, $n = pq$ and $x \in QR(n)$. Let $p = 2r + 1$ and $q = 2s + 1$. Let π be the period of the sequence $x, x^2, x^4, x^8, ..., x$. Then π is the smallest positive integer such that

$$x^{2^\pi} \equiv x \bmod n.$$

The first observation to make is that since $x \in QR(n) \subset \mathbb{Z}_n^*$ then x is relatively prime to n hence

$$x^{2^\pi - 1} \equiv 1 \bmod n.$$

Now since $n = pq$, $x \in QR(p)$ and $x \in QR(q)$, hence

$$x^{2^\pi - 1} \equiv 1 \bmod p.$$

Let α be a primitive element in $\mathbb{Z}_p$. Noting that

$$QR(p) = \{\alpha^2, \alpha^4, ..., \alpha^{p-1} = 1\} = \{\alpha^0 = 1, \alpha^2, ..., \alpha^{p-3}\}$$

it follows that $x = \alpha^{2k}$ for some $0 \leq k < (p-1)/2 = r$. Hence

$$x^{2^\pi - 1} = \alpha^{2k(2^\pi - 1)} \equiv 1 \bmod p.$$

If α is a primitive element in $\mathbb{Z}_p$ then

$$(p-1) | 2k(2^{\pi - 1})$$

and since $(p-1) = 2r$ then

$$r|k(2^\pi - 1).$$

Now suppose r is prime. Then since $0 \leq k < r$ either $k = 0$ or $r|(2^\pi - 1)$. That is either $x = 1$ or

$$2^\pi \equiv 1 \bmod r.$$

Hence the period modulo p is either 1 if $x = 1$ or it depends on the order of 2 modulo r where r is prime.

Let us turn to the particular case in example 2.5 where $p = 7 = 2 \times 3 + 1$ and $q = 11 = 2 \times 5 + 1$. In this case 3 and 5 are prime and we find that the order of 2 in $\mathbb{Z}_3$ is 2 and the order of 2 in $\mathbb{Z}_5$ is 4, that is, 2 is a primitive element in $\mathbb{Z}_3$ and in $\mathbb{Z}_5$. In $\mathbb{Z}_n$ where $n = pq$, if we have cycles of period π_p in $\mathbb{Z}_p$ and π_q in $\mathbb{Z}_q$, then we will have cycles of period $\text{lcm}(\pi_p, \pi_q)$. In our example we have cycles of period 1 and 2 modulo p and 1 and 4 modulo q, hence, since $\text{lcm}(2, 4) = 4$ we can only have cycles of period 1, 2, and 4 modulo $n = 77$. In the data shown in example 2.5 we find cycles of each possible period.

Now back to the problem of how to choose p, q and the seed x such that the cycles in the BBS sequence will be large. If we choose r such that 2 is a primitive element in $\mathbb{Z}_r$ then 2 will have order $r - 1$ hence the period π of $x \in QR(p)$ is $r - 1$. We can do the same for the period modulo q. Hence we have the following solution to the problem:

Choose r prime such that $p = 2r + 1$ is prime and 2 is a primitive element in $\mathbb{Z}_r$. Likewise choose $s \neq r$ prime such that $q = 2s + 1$ is prime and 2 is a primitive element in $\mathbb{Z}_s$. Then we have $x = 1$ has period 1 and any other $x \in QR(n)$, i.e. $1 < x < n$, has period $\pi \in \{r - 1, s - 1, \text{lcm}(r - 1, s - 1)\}$, all of which are large.

Hence choosing the seed x_0 from $QR(n)$ such that $x_0 \neq 1$ will necessarily have a long period. To choose r such that 2 is a primitive element in $\mathbb{Z}_r$ we may choose r and s of the form $2t + 1$ where t is prime so that the order of 2 in $\mathbb{Z}_r$ is one of $2, t, 2t$. Note, it is a consequence of such a choice for p, namely

$$p = 2r + 1 = 2(2t + 1) + 1 = 4t + 3$$

that p will be congruent to 3 modulo 4.

5 A Maple BBS Generator

In this section we construct BBS generators for cryptographic applications using primes of length 512, 768, and 1024 bits. From section 4 the period of these generators will be at least 2^{511}, 2^{767} and 2^{1023} respectively. Because the pseudo-random bits produced by the BBS generator are cryptographically secure, they should pass all statistical tests for randomness. Hence even for primes for which n could be factored, the pseudo-random bits generated by the BBS generator should be very good for non-cryptographic applications. For this reason we also provide BBS generators using the first allowable primes less than 32, 48, and 64 bits in length since these generators will be considerably more efficient than the BBS generators with primes of length 512 bits and longer. At the end of this section we will compare the efficiency of the BBS generators with Maple's `rand` command.

Generating the Primes

In section 4 we showed that if we choose the primes p and q to be of the form $2r+1$ and $2s+1$ respectively where r and s are also prime and 2 is a primitive element in $\mathbb{Z}_r$ and $\mathbb{Z}_s$ respectively, then any $x_0 \in QR(n)$ that is not 1 will have a long period $\pi \in \{r, s, \mathrm{lcm}(r,s)\}$. For 32 bits we obtain the primes $p = 2^{32} - 11129$ and $q = 2^{32} - 65609$. For 48 bits we obtain the primes $p = 2^{48} - 23417$ and $q = 2^{48} - 116489$. For 64 bits we obtain the primes $p = 2^{64} - 704009$ and $q = 2^{64} - 794489$.

If we require that the primes p and q be large enough so that factoring $n = pq$ is computationally infeasible then finding primes of the required form becomes difficult. To find a prime p of this form such that 2 is a primitive element in $\mathbb{Z}_r$ we required that $r = 2t+1$ where t is also prime. The obvious approach to find primes of this form is to generate an odd integer t and test if t is prime, then test if $r = 2t+1$ is prime, then test if $p = 2r+1$ is prime and then test if $2^t \not\equiv 1 \bmod r$. The cost of a Fermat pseudo-prime test, testing if $3^{p-1} \equiv 1 \bmod p$ is $O(\log^3 p)$. From the prime number theorem, the density of primes is $1/\ln p$ hence we expect to find that one random number in $\ln^3 p$ satisfies the above requirements. This means that the cost of the search is $O(\log^6 p)$. It is considerably more efficient to sieve out non-prime t, r, p using trial division. We use trial division of all primes up to 499. Then we apply Fermat pseudo-primes to the base 3. If t, r, p pass these tests then they are very probably prime. Next we test if 2 would be a primitive

element in $\mathbb{Z}_r$. For this it suffices to test if $2^t \not\equiv 1 \bmod r$. Finally we apply Maple's `isprime` command to test the primality of t, r, p. This order of the computation efficiently sieves out all non-prime triples t, r, p before the they are tested for primality. Nevertheless, this took 50,000 seconds in Maple on a 3.0GHz computer to find the 1024 bit primes of the required form. The values of n for 512, 768, and 1024 bit primes are shown in the appendix as the values `n512`, `n768` and `n1024`.

Generating the Random Bits

To build a BBS generator we note the following: in section 3 we showed that the least significant bit of the quadratic residues is cryptographically secure. It can be shown that the least significant $\log_2 \log_2 n$ bits are cryptographically secure. For 512 bit primes, this means we can use the least significant 10 bits, i.e. we get 10 pseudo-random bits per quadratic residue. For the BBS generators with primes of length 32, 48, 64 bits we use more than the least significant $\log_2 \log_2 n$ bits for increased efficiency.

Generating the Seed

For cryptographic purposes, the user should input a random $x_0 \in QR(n)$ such that $x_0 \neq 1$ as the seed of the BBS generator. One way to do this is as follows

Algorithm Generate Seed
repeat
    repeat *choose* $x \in \mathbb{Z}_n$ *at random* **until** *gcd(x, n) = 1.*
    set $x_0 = x^2 \bmod n$.
until $x_0 > 1$.
Output x_0.

To simplify this procedure for non-cryptographic purposes, we square x until it exceeds n. Our Maple procedure below takes three inputs, l the length of the primes in bits, $k > 0$ the number of bits to be output, and the seed. It outputs a procedure, the PRBG, which when called outputs a sequence of L bits. The first input l is given as a subscript. If this is not specified the default $l = 512$ is assumed. We show an example where we use 768 bit primes and generate 60 random bits. The Maple code for the BBS procedure is given in the Appendix.

```
> B := BBS[768](20,2):
> B();
          0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1
> B();
          1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0
> B();
          0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0
```

The following timing results are comparing the time that it takes a k bit BBS generator to generate L bits and with the time it takes Maple's linear congruential PRNG to generate one 10 decimal digit pseudo-random number.

k	L	ratio	L	ratio
32	11	1.2	33	4.6
512	10	13.7	30	36.7
1024	11	53.8	33	160.1

Table 1: Maple BBS generator Timings

Thus generating 11 pseudo-random bits using a 32 bit prime BBS generator takes 20% longer than generating one 10 decimal digit random number using Maple's LCG `rand` and generating 30 pseudo-random bits using a 512 bit prime BBS generator takes over 36 times longer than Maple's LCG.

6 Conclusion

We have presented a Maple implementation of the Blum-Blum-Shub generator and provided the reader with a sketch of the proof that the BBS generator is cryptographically secure. We have shown how to choose the primes so that the input seed cannot accidentally hit a cycle with small period. Our implementation provides the user a choice of prime lengths of size 512, 768, and 1024 bits which offers a range of security at additional computational cost.

We also provide BBS generators for primes of length 32, 48, and 64 bits. While not cryptographically secure, these generators are expected to yield high quality pseudo-random bits at a modestly higher cost than Maple's built-in LCG. This is about the best we can hope to achieve using modular arithmetic because all pseudo-random number generators which are linear functions are not cryptographically secure and the Blum-Blum-Shub generator, which computes $x^2 \bmod n$, is the simplest (cheapest to compute) non-linear function.

References

[1] Knuth, D. E., *The Art of Computer Programming: Volume II Semi-Numerical Algorithms,* second edition, Addison Wesley, 1981.

[2] Stinson, D. R., *Cryptography: Theory and Practice* first edition, CRC Press, 1995.

[3] L. Blum, M. Blum, M Shub, A Simple Unpredictable Random Number Generator. *SIAM Journal on Computing*, **15**, pp 364–383, 1986.

Appendix

The following Maple code may be downloaded from
http://www.cecm.sfu.ca/CAG/products.html

```
n512 :=
260666452939490510307195929055629561353302281074699610439425132699148 42\
48348492923796467962038100608110411889186084921226820413409714311916621 3\
30457037177391969005366566139223496222462773495292813549669189185224249 3\
10134174467129337447068810077317462049525433477860112592306288867332234 0\
4213064415720576165169:

n768 :=
735064535964616210859509956353323918488471699185346493572621478881863 48\
43137164001860306489083958634401769259602392350730546634843222620992962 5\
32712084954402182331364966323999960574035017960401901871761291936520824 9\
98655216324143930637411349012127158049739694262301256759927284741885502 9\
27060784529846537689862998056931897995339505885789987808180874603368507 9\
46186357332824887734109895393160464577736037680127991496487983777234017 5\
30466276827934373481537277639547 3:

n1024 :=
635889556583303470220852308455762122983221277287779039164290758608492 90\
71132845514208579772157672029249844241528488503729099603302068153336229 4\
49475216470037707964501679044251308281941454266979092966822214806472783 2\
66300171107102915404579499623681984877421312176460934436577205556109838 5\
18550998086153009196252383898700986899499776455580901450919422131400647 9\
97816771933577390142325628788963500705655433444972332940782313674275575 1\
86913409549954398028774469503226168640074908181127066257672361045150448 4\
64639538059326242057012423830887510240212368507183232254239727512835259 5\
49723357647718327998754795646911307672321:
```

```
BBS := proc(k::posint,seed::posint) local x,g,T,i,j,l,m,n,q,z,t;

 if type(procname,indexed) then l := op(procname) else l := 512; fi;
 if    l=32 then m := 10; n := 18446414487239353729;
 elif  l=48 then m := 11; n := 79228162474884299504590734913;
 elif  l=64 then m := 12; n := 340282366920910821054273642375418145089;
 elif  l=512 then m := 10; n := n512;
 elif  l=768 then m := 10; n := n768;
 elif l=1024 then m := 11; n := n1024;
 else error "prime bit length must be 32, 48, 64, 512, 768 or 1024";
 fi;

 if seed <= 1 or seed >= n-1 then
    error "seed must be in the range 1 < seed < n-1"; fi;
 # Note the following should never happen; it means that someone
 # ‘‘guessed’’ the factorization of n = p q which is extremely unlikely
 if igcd(seed,n)>1 then seed := seed+1; fi;

 # Don't allow x0 be small e.g. 2, 3, because squaring will
 # preserve the parity of x until x becomes bigger than n.
 # Also, squaring x at least once ensures that x_0 is in QR(x)
 x := seed; while x < n do x := x^2 od; x := irem(x,n);

 T := Array(0..2^m-1); # m random bits at a time
 for i from 0 to 2^m-1 do t := i; T[i] := seq(irem(t,2,'t'), j=1..m) od;

 if k = m then
    subs( 'L' = 2^m, proc() x := irem(x^2,n); T[irem(x,L)]; end );
 else
    z := [0$k];
    subs({'K' = k, 'L'=2^m},
    proc()
        z := z[k+1..-1];
        while nops(z) < k do x := irem(x^2,n); z := [op(z),T[irem(x,L)]] od;
        op(1..k,z);
    end);
 fi;

end:
```