4.1 The BBS Generator

As with the RSA cipher we consider an integer module n that is a product of two large prime numbers. For the BBS generator we choose a Blum **integer**, preferably—but not necessarily—a special or even superspecial one.

> Choosing n superspecial ensures that the sequence of states has a huge period. See the discussion in the Appendices 12–14 of Part III. However the following security proof doesn't depend on this property.

The BBS generator works in the following way that easily fits into the general framework of Figure 2.1: As a first step choose two large random Blum primes p and q, and form their product $n = pq$. The factors p and q are internal (secret) parameters, the product n may be treated as internal or external (public) parameter. As a second step choose a random integer, the "seed", s with $1 \leq s \leq n - 1$, and coprime with n.

> The coprimality is efficiently tested with the Euclidean algorithm. If we catch an s not coprime with n, we have factorized n by hazard. This might happen, but is extremely unlikely, and can easily be captured at initialization time.

Then we proceed with generating a pseudorandom sequence: Take $x_0 = s \in \mathbb{M}_n$ as initial state, and form the sequence of inner states of the pseudorandom generator: $x_i = x_{i-1}^2 \bmod n$ for $i = 1, 2, 3, \ldots$ In each step output the last significant bit of the binary representation, that is $u_i = x_i \bmod 2$ for $i = 1, 2, 3 \ldots$, or in other words, the parity of x_i.

> If $x_i < \sqrt{n}$, then $x_i^2 \bmod n = x_i^2$, the integer square, so x_{i+1}^2 has the same parity as x_i. In order to avoid a constant segment at the beginning of the output, often the boundary areas $s < \sqrt{n}$, as well as $s > n - \sqrt{n}$, are excluded. However if we really choose s as a true random value, the probability for s falling into these boundary areas is extremely low. But to be on the safe side we may require $\sqrt{n} \leq s \leq n - \sqrt{n}$.
>
> If the seed s happens to be a quadratic non-residue, the sequence of inner states (the BBS sequence) has a preperiod of length 1.

Example

Of course an example with small numbers is practically irrelevant, but it illustrates the algorithm: Take $p = 7$, $q = 11$, $n = 77$, $s = 53$. Then $s^2 = 2809$, hence $x_1 = 37$, and $u_1 = 1$ since x_1 is odd. The following table shows the beginning of the sequence of states:

i	1	2	3	4	...
x_i	37	60	58	53	...
u_i	1	0	0	1	...

Since $x_4 = 53 = s$ the seed s happens to be a quadratic residue, and the BBS sequence has period 4. Therefore the output "pseudorandom" sequence (u_i) a forteriori has period 4.

Treating the primes p and q as secret is essential for the security of the BBS generator. They serve for forming n only, afterwards they may even be destroyed—in contrast with RSA there is no further use for them (except when you use SageMath, see below). Likewise all the non-output bits of the inner states x_i must be secret. Moreover there is no reason to reveal the product $n = pq$ even if the following security proof doesn't depend on the nondisclosure of n.

SageMath has an implementation of the BBS generator via the methods `random_blum_prime()` and `blum_blum_shub()`. The code sample 4.1 shows how to use them.

Sage Example 4.1 Generating a pseudorandom bit sequence by the BBS generator

```
sage: from sage.crypto.util import random_blum_prime
sage: from sage.crypto.stream import blum_blum_shub
sage: p = random_blum_prime(2^511, 2^512)
sage: q = random_blum_prime(2^511, 2^512)
sage: s = 11.powermod(248,p*q)  # a (not so random) example
sage: prseq = blum_blum_shub(1024,s,p,q)
```

Table 4.1 shows a Blum integer with 309 decimal places (or 1024 bits) that was an intermediate result of this program. Considering the progress of factoring algorithms we better should use Blum integers of at least 2048 bits.

```
 4506 15286 74466 50249 26225 14044 26383 22616 74480 10227
69340 10344 80414 96318 08671 21639 63710 30387 17602 25696
53909 02080 09976 45161 76261 91025 59480 62175 49124 86394
40823 70452 14981 62658 94574 67753 74945 83135 16199 61782
07594 51105 16833 44889 30109 66289 10763 64987 90309 41852
27681 66632 02722 32988 57145 85172 07427 89442 30004 31819
83739 34537
```

Table 4.1: A 1024 bit Blum integer

Table 4.2 shows the resulting bitsequence. Be warned that the SageMath

output is of type `StringMonoidElement`. For further use in a stream cipher it might be necessary to convert it to a bitblock or bitstring.

```
1000 1111 1001 0101 1001 0111 0011 0100 0010 1000 1100 0001
1010 0101 1110 1001 1010 1001 0110 0010 1010 1010 0111 0111
1000 1010 1000 1101 1111 1101 1010 1100 1100 0001 0101 1001
0111 1111 0001 0100 1010 0000 1100 1010 0101 1000 1110 0000
0001 1011 0100 0100 1010 0010 1010 1010 0110 1001 0111 1100
1011 0010 0011 0100 1101 1001 0101 0100 0111 0100 0010 0111
1101 1000 0010 0111 1000 0110 1110 0111 1110 1101 0110 1000
0001 0011 1111 0011 0011 0101 0001 0001 1010 0110 0101 1000
1010 1100 1011 0011 1111 1000 1001 0100 0001 1110 1111 1111
1001 0000 0010 0000 0111 0111 1001 0001 1111 0100 1010 0011
1000 0111 1100 0000 1011 0110 1011 1010 0111 0100 1110 1001
1001 0101 0011 1000 0010 0011 1010 1001 1100 0010 1111 1001
1010 1001 0110 0011 1001 0100 1000 1111 1001 1001 0010 1000
0111 0110 1101 0011 0110 0010 1110 0010 0000 1100 1011 1111
0011 0010 0110 1110 1000 1000 1110 1110 0011 0010 0100 0100
1101 1000 0011 0010 1000 1110 1000 1101 1010 0001 0011 1100
1001 0110 1010 0000 0000 0000 1011 0111 1010 0010 1100 1010
0100 0010 0010 0010 0010 1011 0100 0000 1100 1010 1101 0000
1101 1111 0011 0001 1000 0000 0111 0111 1110 1111 0011 1011
1111 0001 0010 1000 0110 1011 0111 0011 1111 1011 0101 0100
0110 1111 1111 0011 1011 0000 1010 0010 1100 0010 1001 0101
1110 1001 1001 1001
```

Table 4.2: 1024 "perfect" pseudorandom bits. Note that generating 1024 pseudorandom bits from a 1024-bit random integer isn't worth the effort. However we could continue this sequence much further and generate, say, 2^{30} pseudorandom bits.

Figure 4.2 gives an optical impression of the randomness of this sequence, and Figure 4.3, of its linearity profile.

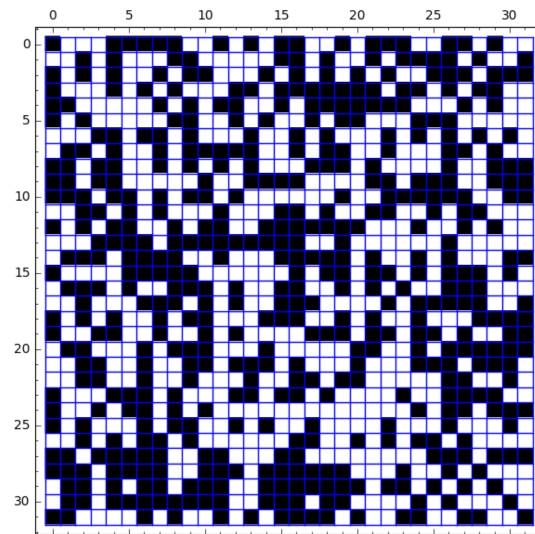

Figure 4.2: Visualization of a “perfect” pseudorandom sequence

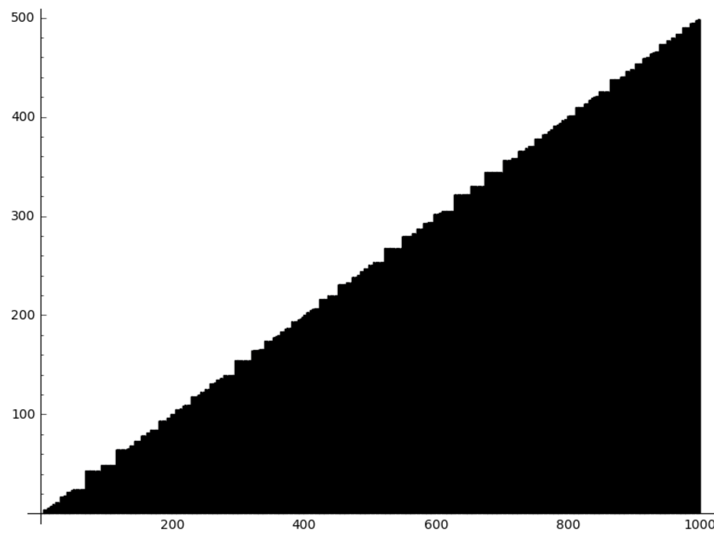

Figure 4.3: Linearity profile of a “perfect” pseudorandom sequence