About Random Bits

Martin Geisler ⟨mg@daimi.au.dk⟩
Mikkel Krøigård ⟨mk@daimi.au.dk⟩
Andreas Danielsen ⟨beldin@daimi.au.dk⟩

December 3, 2004

Contents

Introduction

Random numbers (and therefore also random bits) are becoming increasingly important in a number of settings as strong cryptography finds use in more and more applications.

If an application wants to do strong cryptography, then it does not matter how good the cryptographic algorithms are if the seeds used to initialize them can be effectively predicted. Over the years there have been several examples of software implementations where the seeds used were weak. A prime example is the Netscape browser, which in an early version used the system clock to derive a seed for use with SSL connections [3]. When such a connection was made, an adversary would have a good estimate of the system time, and would therefore end up with a small set of candidates for the seed.

We will start by looking at pseudo-random bit generators in general. Section 2 describes how to use physical processes to obtain random bits, and how they can be turned into secure random bits. We present our implementation of a practical way of obtaining secure random bits in Section 3. Although practical, the generation will still be too slow for most uses, so Section 4 presents the Blum Blum Shub (BBS) algorithm for generating many random bits from only a small random input. We will also give a proof of security for BBS, which is based on an assumption about the hardness of a certain number-theoretic problem. Finally, in Section 5 we will test our generators using a statistic test-suite.

1 Random Bit Generators

In this section we will define several kinds of bit generators and explain what it means for such a generator to be cryptographically strong. Random number generators are easily constructed using random bit generators, so it is enough to look at bit generators. First we need to be able to measure the "randomness" of a bit.

Definition 1. Let b be a bit which takes the value 0 with probability p, and 1 with probability $1 - p$. The *bias* ε is then $\varepsilon = p - \frac{1}{2}$. We say that a bit with a bias $\varepsilon = 0$ is *unbiased*.

We are now ready to state the properties we expect from a good random bit generator.

Definition 2. A *true random bit generator* is something that produces independent and unbiased bits.

1.1 Pseudo-Random Bit Generators

In Section 3 we will give an example of a true random bit generator. Unfortunately, true random bit generators are generally slow and may consist of special hardware. Therefore we need generators that produce bits faster and do it in software.

Definition 3. A *pseudo-random bit generator* (PRBG) is a deterministic algorithm that given some initial value, called the *seed*, produces a number of random bits.

Since they are deterministic, PRBGs always produce the same bits for each seed. For this reason, it is obvious that a PRBG cannot be secure if the seed is predictable in any way. The idea is to use a true random bit generator to make the seed, which a PRBG will use to generate a much larger number of bits — this is what we call *boosting randomness*. In cryptography, we will need certain high-quality PRBGs so that it is computationally infeasible to predict the generated numbers. There are two definitions of what it means for a PRBG to be cryptographically strong: passing all *polynomial-time statistic tests* and passing the *next-bit test*. According to Yao [18] these definitions are equivalent, so we will be using the next-bit test. Moreover, it does not matter from the point of view of the statistical tests if the sequence is reversed. So predicting the next or previous bit are equally hard problems.

The set-up for the next-bit test is the following: We have an adversary $\mathcal{A}$ and an oracle $\mathcal{O}$. The oracle knows a bit sequence s, and $\mathcal{A}$ is allowed to ask the oracle about the next bit any number of times, as long as there is at least one unseen bit left. After seeing the first l bits, $\mathcal{A}$ must guess the $l+1$'st bit. $\mathcal{A}$ wins if this guess is correct. We define the advantage of $\mathcal{A}$ to be $\text{adv}_{\mathcal{A}} = |1/2 - p_{\mathcal{A}}|$, where $p_{\mathcal{A}}$ is the probability that $\mathcal{A}$ succeeds.

Definition 4. A PRBG is said to be *cryptographically strong* if it passes the next-bit test. That means that for any polynomial-time adversary $\mathcal{A}$, $\text{adv}_{\mathcal{A}}$ is negligible. The time and advantage are functions of a security parameter.

So if a PRBG is cryptographically strong, an adversary $\mathcal{A}$ might as well flip a coin to predict the next bit. The security parameter mentioned above could be a bit length, for example the security parameter we have in BBS is the bit length of a number $n = pq$ where p and q are (Blum) primes, see Definition 10 on page 9.

2 Physical Sources for Random Bits

The best way to obtain truly random bits would be to observe a physical phenomenon which is believed to exhibit random behavior. The basic postulates of Quantum Mechanics tells us that one cannot reliably predict the outcome of certain physical experiments: a well-known example is the question of when a radioactive atom will decay.

Unfortunately one does not find radioactive sources and Geiger counters as a standard peripheral on modern PCs, so this way of obtaining random bits is not practical. That is not to say that it has not been done, see HotBits by Walker [13]. Other examples of exotic sources of randomness includes `random.org` by Haahr [4] and the LavaRnd project by Noll et al. [11].

2.1 Exploiting Air Turbulence in Hard Disk Drives

Another physical phenomenon which is believed to be unpredictable is *air turbulence*. With our current understanding of the underlying dynamics, we

cannot predict the behavior of turbulent air to any high degree. Likewise, objects moving in air will both cause and be influenced by the turbulence, and so they are believed to be a good source of unpredictable behavior [2].

Fortunately we have such objects on all modern PCs: the hard disk drive, see Appendix A for a description of the operation of a hard disk drive. Studies made by Davis et al. [2] on big models of disk drives submerged in water have shown that there is a strong turbulence present when a drive operates. This turbulence is expected to disturb the time it takes to repeatedly read the same block from the disk, and the access times can therefore be used as a source of random bits.

Jakobsson et al. [7] describes how they successfully implemented a program to extract random bits from timed readings on a hard disk drive. We have made a program which follows their basic ideas, but which deviates in a number of ways.[1] This is the topic of Section 3. Here we will now describe how the readings from any source can be transformed into unbiased bits.

2.2 Generating Unbiased Random Bits

If one were to take the raw readings and convert them into bits directly, then one would most likely not end up with a true random bit generator as defined by Definition 2. The problem is that we have no reason to believe that the bits are uncorrelated and unbiased.

Theorem 6 gives us a way to transform any sequence of biased bits into a single bit with an arbitrary small bias. To prove this we need the following lemma.

Lemma 5. *Given a sequence of independent bits $b_1, \ldots, b_n$ with corresponding biases $\varepsilon_1, \ldots, \varepsilon_n$, then the bit b defined by*

$$b = \bigoplus_{i=1}^{n} b_i$$

has bias

$$\varepsilon = 2^{n-1} \prod_{i=1}^{n} \varepsilon_i.$$

Proof. We prove the claim by induction in n, the number of bits. With only a single bit the result clearly holds. Now assume it holds for n bits so

$$b = \bigoplus_{i=1}^{n} b_i \quad \text{has bias} \quad \varepsilon = 2^{n-1} \prod_{i=1}^{n} \varepsilon_i.$$

We will show the result for $n + 1$ bits. We start by writing

$$\bigoplus_{i=1}^{n+1} b_i = b \oplus b_{n+1}.$$

[1] It has come to our attention that our work might be illegal(!) — see United States Patent No. 6,317,499 [5]. But since Davis et al. [2] state that the idea of extracting random bits from storage devices is more than 50 years old (citing Lewis [9], itself an old book from 1975), we cannot really take the patent seriously.

By the induction hypothesis the bias of b is ε, so the probability of b being 0 is $\varepsilon + \frac{1}{2}$. Similarly b_{n+1} is 0 with probability $\varepsilon_{n+1} + \frac{1}{2}$. So we have

$$\begin{aligned}\Pr[b \oplus b_{n+1} = 0] &= \Pr[b = 0 \wedge b_{n+1} = 0] + \Pr[b = 1 \wedge b_{n+1} = 1] \\ &= (\varepsilon + \tfrac{1}{2})(\varepsilon_{n+1} + \tfrac{1}{2}) + (\tfrac{1}{2} - \varepsilon)(\tfrac{1}{2} - \varepsilon_{n+1}) \\ &= \frac{1}{2} + 2\varepsilon\varepsilon_{n+1} = \frac{1}{2} + 2^n \prod_{i=1}^{n+1} \varepsilon_i.\end{aligned}$$

Thus the bias of $n+1$ bits $b_1, \ldots, b_{n+1}$ is $2^n \prod_{i=1}^{n+1} \varepsilon_i$, which was what we wanted to prove. □

Theorem 6. *Given $\varepsilon > 0$ and a sequence $b_1, \ldots$ of bits with biases $\varepsilon_1, \ldots$ there exists an n such that*

$$b = \bigoplus_{i=1}^{n} b_i$$

has bias less than ε.

Proof. This follows from Lemma 5 by writing

$$\varepsilon = 2^{n-1} \prod_{i=1}^{n} \varepsilon_i = \frac{1}{2} \prod_{i=1}^{n} (2\varepsilon_i).$$

Since $\varepsilon_i < \frac{1}{2}$ for all i we have that $\varepsilon \longrightarrow 0$ for $n \longrightarrow \infty$. □

3 Generating Random Seeds

In this section we will describe our program which uses timings from a hard disk drive to derive what we assume to be true random bits. The program works as follows:

1. Temporary files are created, one which will be used for the timings (called the work file), and one which will be used to flood the disk cache on the drive (the disk file).
2. A sequence of good offsets into the work file is determined. Subsection 3.2 describes what *good* offsets are, and how they are found.
3. A number of timing runs are made through the sequence of offsets.
4. The timing results are reduced to single bits, and an estimate of the bias is calculated.
5. Using Theorem 6 we then calculate the number of reduced timings that need to be combined (using exclusive-or) to produce a single timing with a known small bound on the bias.
6. The program is now ready to produce bits: we repeatedly run through the offsets, combine the timings to lower the bias, output a bit (or byte) and repeat.

3.1 Obtaining Random Timings

As described in Section 2, we expect air turbulence inside the disk drive to disturb the time it takes to read a given block from the disk. This disturbance is rather small, but still measurable — we just have to make sure that our read requests really result in a physical read on the disk.

The key is to defeat the buffering in the OS and the disk cache. Since hard disk access is slow compared to main memory access, the OS and the hard disk try to buffer data as much as possible. This involves storing recently read data, and reading ahead in an attempt to predict future read requests.

The buffering on the OS level can easily be turned off (on a Linux system) by opening a file with a `O_DIRECT` flag set. This can be done without needing any special privileges.

Bypassing the hard disk cache is a more difficult problem. Modern hard disks typically have a cache 8 MiB, but it is not sufficient to simply read 8 MiB of random data to fill up the cache. This is a result of *cache segments* which make the cache acts as if it consisted of several independent caches. So with four segments, the cache would be able to intelligently cache the data read by four programs at four different places on the disk.

So if our program simply reads 8 MiB of data we would most likely just fill one segment. The answer is to read from four different places in the disk file, and so convince the disk that it should cache the data in different segments. The result is that the entire cache is filled with our garbage bytes, and so our next read request to the work file will result in a physical read.

3.2 Locating Good Offsets

The aim of the program is to output highly random bits as fast as possible, so we want the program to take advantage of as much available entropy in our source as possible. This is done by selecting offsets in the file that corresponds to large access times.

As described in more detail in Appendix A, the time taken to access a particular block on a hard disk is the sum of several factors, including seek time and rotational latency. We are interested in access times with as large a rotational latency as possible, for it is this factor that will render our measurements unpredictable. The other factors, such as interrupts, could (in principle) be predicted by an adversary, but we believe that the turbulent air inside the disk is unpredictable.

Finding such offsets with a large rotational latency boils down to finding offsets with a large access time — after all, that is the only thing we can measure from the program. Fortunately it turns out that one can measure the rotational latency quite easily by simply timing blocks further and further into file, for this means that we time further and further along a track. The program measures the time it takes to read blocks at offsets a and b, for a fixed a and increasingly large b.

The access times increases (almost) linearly up to a maximum, followed by a sudden drop, see Figure 1 on the next page. The drop occurs when we have read all the way around a track on the disk, something which can be seen from the fact that the access time (for a 7200 RPM disk) drop with just

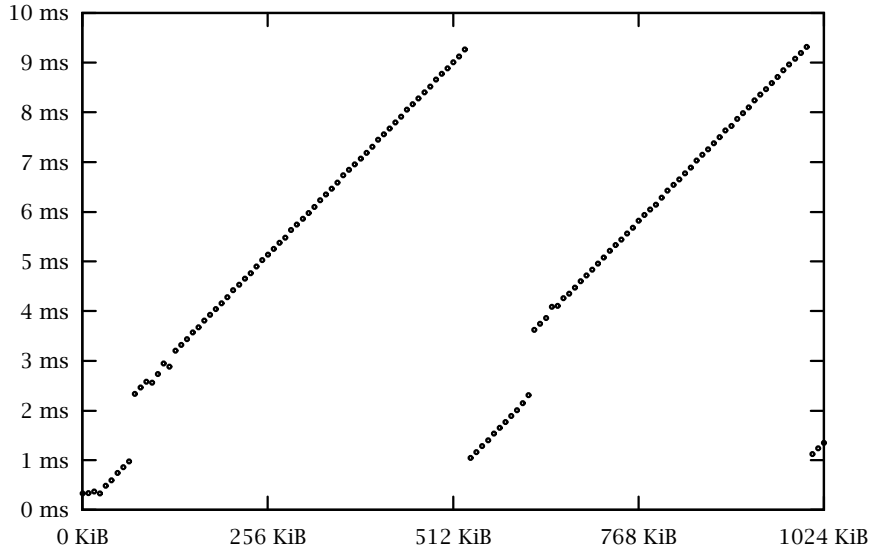

Figure 1: Access times versus file offset. From offset 528 KiB to 536 KiB there was a drop of 8.2 ms, which tells us that this was made on a 7200 RPM disk.

around 8.3 ms, which is the time it takes for such a disk to make one revolution. As seen in Figure 1 the increase is not completely linear, where the small jumps observed are probably due to fragmentation in the hard disk drive, see Appendix A. These jumps can result in a pair of addresses that does not require a full rotation, and therefore does not give the maximal entropy. The jumps are rather small and we believe that, even in a worst case scenario, the loss in entropy does not corrupt our final result.

3.3 Estimating the Bias

Given a good set of offsets we need to estimate the bias of the timings. First each timing is turned into a *reduced timing* by taking the exclusive-or of all the bits in the timing. The single resulting bit will be 0 or 1, and these bits will be independent since they corresponds to independent timings on the disk. The hope is that the fluctuations in the timings will give us a fairly unbiased source of bits in this way.

Our program estimates the bias by counting the number of zero bits, this directly gives us p, the probability of getting a 0. From p we find the bias ε as $\varepsilon = |p - 1/2|$, thereby treating a negative bias as a positive. This works because of the symmetry present in all the calculations: if $p < \frac{1}{2}$, then our bias corresponds to having p be the probability of getting a 1 instead of a 0.

Assuming that all the readings from the disk have the same bias, we can use Theorem 6 to drive the bias down under any user-specifiable bound. Letting $\hat{\varepsilon}$ denote the bias just calculated and solving for n, the number of

bits to combine, we obtain the expression

$$n = \frac{\log_2 \varepsilon - 1}{1 + \log_2 \hat{\varepsilon}},$$

where ε is the desired bound on the bias. With a default setting of 2^{-20} for the maximum allowed bias, our program normally needs to combine between 4 and 8 reduced timings to ensure the low bias.

Notice that this calculation ensures a high entropy in the output, since a low bias gives a probability close to $\frac{1}{2}$, which in turn gives a high entropy.

3.4 Harvesting the Entropy

With a knowledge of the available entropy, our program is now finally ready to do what it was created for: outputting highly random bits.

This step is fairly simple, consisting only of a never-ending loop in which the program runs through the sequence of offsets determined earlier. The timings are reduced, and the reduced timings are combined into single bits. The bits are then combined into bytes which are outputted on the standard output stream.

4 Boosting Randomness

Given a small number of random bits, it is possible to generate a much larger sequence of bits that are computationally infeasible to predict. In fact, it has been proved that if one-way functions exist, it is possible to construct a cryptographically strong PRBG (CPRBG) [6]. Several CPRBGs have been constructed based on one-way functions such as RSA, and some are based on the hardness of the discrete log problem. The generator we have chosen is based on modular squarings of a number, and prediction of bits can be reduced to deciding quadratic residuosity, which we will discuss below. This generator is quite effective compared with other generators such as the RSA generator, since modular squarings are much faster to perform than larger exponentiations of numbers.

4.1 The Blum Blum Shub Algorithm

The Blum Blum Shub algorithm boosts randomness by using modular squarings and extracting a certain number of the lower-order bits per squaring modulo a number n. The algorithm we study only extracts the lowest order bit, the parity of the number. In the following, we shall describe the algorithm in detail, the theory on which it relies and prove that it is cryptographically strong based on a certain number theoretic assumption. The algorithm is taken from the original paper by Blum et al. [1] and many of the number theoretical results are taken from the notes on quadratic residues in the paper by Junod [8].

To understand the Blum Blum Shub algorithm and its proof of security, we need some basic theory about quadratic residues.

Generate random Blum primes p, q such that $p \neq q$
$n := pq$
Generate random number $s \in \mathbb{Z}_n^*$
$x := s^2 \bmod n$
for $i = 0, \ldots$ **do**
  $x := x^2 \bmod n$
  $b_i := x \bmod 2$
  Output bit b_i
end for

Figure 2: The Blum Blum Shub Algorithm

Definition 7. An integer $x \in \mathbb{Z}_n^*$ is called a *quadratic residue* modulo n if there exists some y such that $y^2 \bmod n = x$. Otherwise, x is a *quadratic non-residue* modulo n. We denote the set of quadratic residues modulo n by QR_n, and the set of quadratic non-residues modulo n by QNR_n.

The following result tells us how $\mathbb{Z}_n^*$ can be divided into certain important subsets.

Theorem 8. *Let $n = pq$ be the product of two distinct odd primes and let $\mathbb{Z}_n^*(+1)$ denote the numbers in $\mathbb{Z}_n^*$ with Jacobi symbol 1 and $\mathbb{Z}_n^*(-1)$ the numbers with Jacobi symbol -1. Then half of the elements of $\mathbb{Z}_n^*$ are in $\mathbb{Z}_n^*(+1)$ and the other half in $\mathbb{Z}_n^*(-1)$. Half the elements of $\mathbb{Z}_n^*(+1)$ and none of the elements of $\mathbb{Z}_n^*(-1)$ are quadratic residues, so $QR_n \subset \mathbb{Z}_n^*(+1)$.*

Now for x chosen uniformly from $\mathbb{Z}_n^*(+1)$, the probability that $x \in QR_n$ is $\frac{1}{2}$. The *quadratic residuosity problem* is the problem of deciding whether such an x is a quadratic residue. Solving this problem is assumed to be hard in the following sense.

Assumption 9. *Any polynomial-time (probabilistic) algorithm $\mathcal{P}$ for deciding the quadratic residuosity of $x \in \mathbb{Z}_n^*(+1)$, where $n = pq$ is the product of two distinct odd primes, will have probability of success at most $\frac{1}{2} + \varepsilon$, where ε is negligible in the bit-length of n.*

This assumption is often called the Quadratic Residuosity Assumption, and we will refer to it by QRA. From now on we will be working with a certain kind of prime called a Blum prime.

Definition 10. A prime p is called a *Blum prime* if

$$p \equiv 3 \pmod 4.$$

We are now ready to look at the BBS algorithm. The pseudo-code for the algorithm is shown in Figure 2. Notice that when s is squared in the algorithm, we will get a quadratic residue modulo n. To prove that BBS is secure, we will look at another problem, which is based on a result which we will not prove here.

Theorem 11. *Let $n = pq$ be the product of two distinct Blum primes. Then every quadratic residue x modulo n has four distinct square-roots. Exactly one of them is also a quadratic residue and we let $\sqrt{x}$ denote this unique root.*

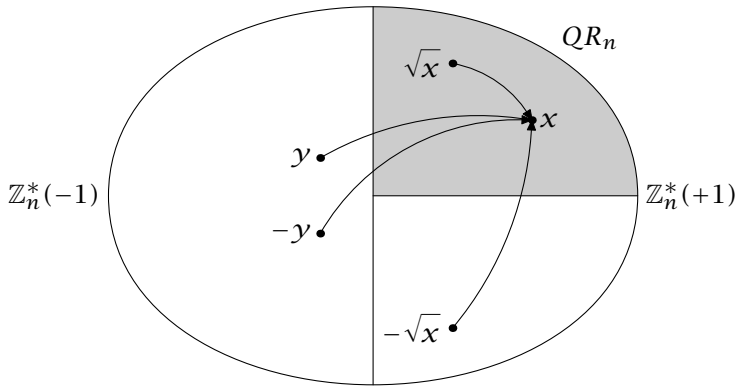

Figure 3: A symbolic map of $\mathbb{Z}_n^*$.

Figure 3 illustrates the four roots of x and their relative location in $\mathbb{Z}_n^*$. The theorem raises a new question: given $x \in QR_n$, what is the parity of the unique root $\sqrt{x}$? Using this problem, we can now show the basic reduction that proves the security of BBS.

Let us assume that we have an algorithm $\mathcal{A}$ which, given a BBS sequence $b_0, \ldots, b_i$, outputs b_{-1} with a certain probability. In other words it predicts sequences to the left. Given a quadratic residue x, we can generate the sequence $b_0, \ldots, b_i$ as in the algorithm and feed this sequence to $\mathcal{A}$. Then $\mathcal{A}$ will find $b_{-1} = \text{parity}(\sqrt{x})$ with the same success probability as before. Let us call this new algorithm $\mathcal{A}'$. But now Theorem 17, which we will prove shortly, tells us that we can build an algorithm $\mathcal{B}$ deciding quadratic residuosity with the same probability of success, and $\mathcal{B}$ runs in polynomial time if $\mathcal{A}'$ does.

What this means is that if a polynomial-time algorithm $\mathcal{A}$ could predict sequences to the left with probability $\frac{1}{2} + \delta$, where δ is non-negligible, there is an algorithm $\mathcal{B}$ deciding quadratic residuosity with probability $\frac{1}{2} + \delta$, which contradicts the QRA. This proves that BBS is a cryptographically strong PRBG.

We need to prove the reduction from the problem of deciding the parity of $\sqrt{x}$ to the problem of deciding quadratic residuosity. To do this we need a little more number theory. The following lemma is based on the isomorphism provided by the Chinese Remainder Theorem.

Lemma 12. *Let $n = pq$ be the product of two distinct odd primes, then*

$$x \in QR_n \iff x \bmod p \in QR_p \wedge x \bmod q \in QR_q.$$

Lemma 13. *Given an odd prime p, then*

$$p \equiv 3 \pmod 4 \iff -1 \in QNR_p.$$

Lemma 14. *Let $n = pq$ be the product of two distinct Blum primes. Then x and $-x$ have the same Jacobi symbol.*

Proof. This follows directly since

$$\left(\frac{-x}{n}\right) = \left(\frac{-1}{n}\right)\left(\frac{x}{n}\right) = \left(\frac{-1}{p}\right)\left(\frac{-1}{q}\right)\left(\frac{x}{n}\right) = (-1)^2\left(\frac{x}{n}\right) = \left(\frac{x}{n}\right).$$

Here we used Lemma 13. □

We now prove that the four roots of x in the symbolic map given in Figure 3 on the preceding page are placed correctly.

Lemma 15. *The function $x \mapsto x^2$ is a 2-1 function on $\mathbb{Z}_n^*(+1)$ when $n = pq$ is the product of two Blum primes.*

Proof. We know that one of the four roots (the one denoted by $\sqrt{x}$) lies in $\mathbb{Z}_n^*(+1)$. Then $-\sqrt{x} \in \mathbb{Z}_n^*(+1)$ as well by Lemma 14. Now assume that a third root $y \in \mathbb{Z}_n^*(+1)$. Then

$$1 = \left(\frac{y}{n}\right) = \left(\frac{y}{p}\right)\left(\frac{y}{q}\right),$$

and we conclude that the two Legendre symbols are equal. But since y is not a quadratic residue, Lemma 12 implies that both symbols are -1. And so $-y$ is a quadratic residue, since

$$\left(\frac{-y}{p}\right) = \left(\frac{-1}{p}\right)\left(\frac{y}{p}\right) = (-1)^2 = 1,$$

and likewise for $(-y/q)$. But this contradicts the fact that $\sqrt{x}$ is the unique root in QR_n.

So for any x exactly *two* of its four roots lie in $\mathbb{Z}_n^*(+1)$, and this proves that the function $x \mapsto x^2$ is 2-1 on $\mathbb{Z}_n^*(+1)$. □

The following lemma more or less directly gives us the reduction we need.

Lemma 16. *Let $n = pq$ where p and q are Blum primes. We have for all $x \in \mathbb{Z}_n^*(+1)$ that*

$$x \in QR_n \iff \text{parity}(x) = \text{parity}(\sqrt{x^2}).$$

Proof. The first case when $x \in QR_n$ is trivial, since Theorem 11 directly tells us that $\sqrt{x^2} = x$.

For the other case take $x \notin QR_n$. Since $(x/n) = 1$ it follows, as in the proof of Lemma 15, that $-x$ is a quadratic residue. So $-x = \sqrt{x^2}$, and x and $-x$ have different parities since n is odd. □

We now prove the main theorem, which basically states that finding the parity of $\sqrt{x}$ is just as hard as deciding quadratic residuosity.

Theorem 17. *Given an algorithm $\mathcal{A}$ that finds the parity of $\sqrt{x}$, we can construct another algorithm $\mathcal{B}$ that decides the quadratic residuosity of x. The two algorithms have the same probability of success.*

Proof. Given $\mathcal{A}$ we can define $\mathcal{B}$ as follows:

$$\mathcal{B}(n, x) = \mathcal{A}(n, x^2 \bmod n) \oplus \text{parity}(x) \oplus 1.$$

Note that the two algorithms output either 1 or 0, where 1 means "yes" and 0 means "no". We want to show that the probability of success for $\mathcal{B}$ is equal to the one for $\mathcal{A}$, that is

$$\begin{aligned}&\Pr[\mathcal{A}(n,x) = 1 \mid G \longrightarrow n, x \in_R QR_n, r_A \in \{0,1\}^{t_A}]\\ &\quad = \Pr[\mathcal{B}(n,x) = 1 \mid G \longrightarrow n, x \in_R \mathbb{Z}_n^*(+1), r_B \in \{0,1\}^{t_B}].\end{aligned}$$

The first probability is the success probability of $\mathcal{A}$. We condition on all the variables that take part in the evaluation of $\mathcal{A}(n,x)$, that is the generation of n, the uniform selection of $x \in QR_n$, and the random choices r_A that $\mathcal{A}$ might need. Similarly for the second probability which is the success probability of $\mathcal{B}$.

First we notice that $t_A = t_B$, that is the two algorithms use the same number of random choices, since $\mathcal{B}$ does not contribute with any extra random choices.

Since the input x for $\mathcal{B}$ is chosen uniformly in $\mathbb{Z}_n^*(+1)$, the input x^2 to $\mathcal{A}$ will also be uniformly distributed in QR_n. This is because the function $x \longmapsto x^2$ is a 2–1 function on $\mathbb{Z}_n^*(+1)$ by Lemma 15.

Lemma 16 says that $\text{parity}(\sqrt{x^2}) = \text{parity}(x)$ if and only if $x \in QR_n$ and so $\mathcal{B}$ guesses correctly exactly when $\mathcal{A}$ does. □

This concludes the proof that BBS is secure. The next section tells us that we can actually do a little better.

4.2 Improvements

It has been proved that predicting BBS sequences to the left is not only as hard as deciding quadratic residuosity, but also as hard as factoring n [12]. But an even stronger result has also been proved — it is possible to extract as much as $\log\log n$ lower-order bits for each squaring. This is exactly what our implementation of BBS does.

5 Statistical Tests

It is impossible to give a mathematical proof that a generator is a truly random bit generator, but it is possible, however, to measure the quality of a random bit generator. This can be done in many ways, for example it is obvious that any good random generator should generate approximately the same amount of zeros and ones. If we were only to test that our sequences consist of equally many ones and zeros, then sequences of the form $0101\ldots$ would be "random", while being highly structured. One could then expand the test by considering so-called *runs* of bits, i.e., subsequences containing the same bit-value. A run of zeros is called a *gap* and a run of ones is called a *block*. We expect a large number of small runs and a small number of larger runs, and we also expect the number of blocks to be approximately equal to the number of gaps. Again one could think of many sequences that obey these rules, but still are not random.

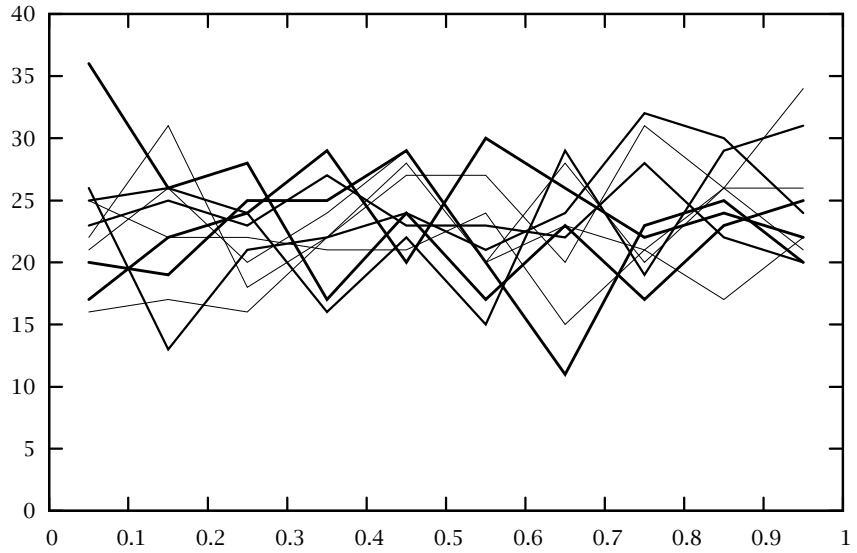

Figure 4: Results using ten 12 MiB files generated using our program to seed.

5.1 The Diehard Test-Suite

To test our random bit generators we have used the so-called "diehard" test-suite[10] which consists of 15 tests. Most of the tests give as result some p-values which, for a random sequence, should be uniformly distributed over $[0, 1)$.

We have tested 4 sequences with diehard: two sequences produced by our BBS random bit generator seeded using Java's built-in random bit generator, and two sequences where our physical random generator produced the seed. The seed given to BBS in both cases is two Blum primes each of length 512 bits, and a random number s of length 1024 bits. The diehard battery of tests only works on files at least of size 11 MiB. Since our true random bit generator only produces 2 bytes per second, it has not been possible to test a random sequence generated only by our true random generator.

We have made a frequency analysis of the obtained p-values, and the results can be seen in Figures 4– 5 on the next page. Each test run gives about 230 values for p, and we made ten tests for each of the two generators. So each line in the plots corresponds to the frequencies found for one test run, and there are ten lines in each plot.

We then grouped the results into buckets of size 0.1 — the idea is that if the p-values are uniform, then each bucket should receive an an approximately equal number of values. That would in turn give flat curves in the plot.

We find it hard to conclude anything based on these figures, other than the fact that both generators produce p-values which appear to be spread

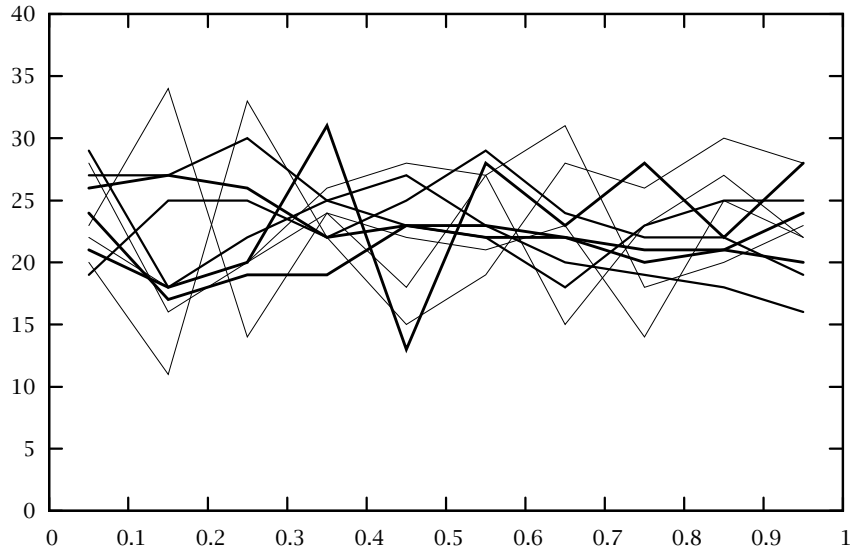

Figure 5: Results using ten 12 MiB files generated using our Java to seed.

uniformly in the interval $[0,1]$. So according to the Diehard test-suite one cannot tell the difference between using the built-in random generator, and our random generator. Of course, if one knows the seed used for the built-in Java random bit generator, then one could predict the seed given by it to the BBS algorithm, whereas there are no seeds involved in the generation of bits from our generator.

6 Conclusion

We have given basic definitions and results regarding random bit generators. An example of a random bit generator which used air turbulence in hard disk drives was presented and implemented. We believe it can be thought of as a true random bit generator, since it is unknown how to predict the effects of air turbulence.

We also gave an example of a cryptographically strong random bit generator (the BBS or $x^2 \bmod N$ generator) which takes true random bits as a seed and boosts them into many more random bits, and we gave a a proof of the security of this generator.

We then implemented and combined the two generators and tested them using the diehard test-suite. In the implementation of the true random bit generator, it is assumed that it is run in an environment without special hardware or privileges.

The results of the Diehard test-suite indicate that our generator is at least as strong as the built-in Java random bit generator, and so it seems to be sound.

A An Introduction to Hard Disk Drives

The following is a summary of the information in [14–17].

A hard disk drive contains a number of rotating *platters* (typically between 1 and 3). In-between the platters one finds the *read-write heads*. During the operations of the drive the heads will move back and forth between the outer edge and the center of the platters (called *seeking*) and this movement will cause turbulence in the air inside the disk. The time used to move the head from one track to another is called the *seek time*, the additional time needed until the wanted data is situated under the head is called the *rotational delay*.

There exists some factors, interrupts and bus arbitration, that affect the time spent from when a user issues a request to the hard drive, until it is performed by the hard drive. The computer bus transfers data and addresses back and forth between the devices to the CPU and main memory. If several programs are running, some of them might try to access the bus at the same time, and then some will experience a delay from when the request is issued until it is actually performed. Interrupts are used by the operating system in multitasking. Since a desktop, with one processor, can only perform one instruction at a time, interrupts are used to stop one program and (re)start another. Since the modern CPUs are so fast, it looks like, to any user, that several programs are now running concurrently.

Fragmentation of the hard disk drive is something that happens when different sized files are created and deleted. When a file is removed it leaves a gap somewhere in the hard disk, and at some point the computer needs to use this space again for a new file. If the new file is larger than the gaps created earlier, the operating system splits the file up into smaller fragments, and writes it to different locations on the disk drive.

B Auxiliary Definitions

In this section we define the Jacobi symbol for the readers who are unfamiliar with it.

Definition 18. Let p be an odd prime. For $a \in \mathbb{Z}_p^*$ the *Legendre symbol* is defined by:

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & p \mid a \\ 1, & a \in QR_p \\ -1, & a \in QNR_p. \end{cases}$$

Definition 19. Let n be odd integer with factorization

$$n = \prod_i p_i^{e_i}.$$

For $a \in \mathbb{Z}_p^*$ we define the *Jacobi symbol* to be

$$\left(\frac{a}{n}\right) = \prod_i \left(\frac{a}{p_i}\right)^{e_i},$$

where (a/p_i) is the Legendre symbol.

References

[1] Lenore Blum, Manual Blum, and Michael Shub. A Simple Unpredictable Pseudo-Random Number Generator. *SIAM Journal on Computing*, 15(2):364–383, May 1986.

[2] Don Davis, Ross Ihaka, and Philip Fenstermacher. Cryptographic Randomness from Air Turbulence in Disk Drives. In *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, pages 114–120. Springer-Verlag, 1994. ISBN 3-540-58333-5.

[3] Ian Goldberg and David Wagner. Randomness and the Netscape Browser. How secure is the World Wide Web? *Dr. Dobb's Journal*, January 1996. Online: `http://www.ddj.com/documents/s=965/ddj9601h/`.

[4] Mads Haahr. `random.org` — True Random Number Service. October 1998. Online: `http://random.org/`.

[5] Bruce Kenneth Hillyer, Bjorn Markus Jakobsson, and Elizabeth Shriver. Storage device random bit generator. United States Patent No. 6,317,499, August 1998.

[6] J. Håstad, R. Impagliazzo, L. Levin, and M. Luby. A Pseudo-Random Generator from any One-Way Function. *SIAM Journal of Computing*, 28(4):1364–1396, 1999.

[7] Markus Jakobsson, Elizabeth Shriver, Bruce K. Hillyer, and Ari Juels. A Practical Secure Physical Random Bit Generator. In M. Reiter, editor, *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 103–111. ACM Press, 1998. ISBN 1-58113-007-4.

[8] Pascal Junod. Cryptographic Secure Pseudo-Random Bits Generation: The Blum-Blum-Shub Generator. August 1999. Online: `http://crypto.junod.info/bbs.pdf`.

[9] T. G. Lewis. *Distribution Sampling for Computer Simulation*. Lexington Books, Lexington, Massachusetts, 1975. ISBN 0-669-97139-1.

[10] George Marsaglia. Diehard. A Battery of Tests for Random Number Generators. January 1997. Online: `http://stat.fsu.edu/~geo/diehard.html`.

[11] Landon Curt Noll, Simon Cooper, and Mel Pleasant. LavaRnd. 2000. Online: `http://lavarnd.org/`.

[12] Umesh V. Vazirani and Vijay V. Vazirani. Efficient and Secure Pseudo-Random Number Generation. In *Proceedings of Symposium on the Foundations of Computer Science*. 1984.

[13] John Walker. HotBits: Genuine Random Numbers, Generated by Radioactive Decay. 1996. Online: `http://www.fourmilab.ch/hotbits/`.

[14] Wikipedia. Fragmentation. In *Wikipedia, the free encyclopedia*. The Wikipedia Community, November 2004. Online: `http://en.wikipedia.org/wiki/Fragmentation`.

[15] Wikipedia. Hard disk. In *Wikipedia, the free encyclopedia*. The Wikipedia Community, November 2004. Online: `http://wikipedia.org/wiki/Hard_disk`.

[16] Wikipedia. Rotational delay. In *Wikipedia, the free encyclopedia*. The Wikipedia Community, June 2004. Online: `http://wikipedia.org/wiki/Rotational_delay`.

[17] Wikipedia. Seek time. In *Wikipedia, the free encyclopedia*. The Wikipedia Community, August 2004. Online: `http://wikipedia.org/wiki/Seek_time`.

[18] A. C. Yao. Theory and application of trapdoor function. In *Proceedings. 23rd IEEE Symp. on Foundations of Comp. Science*, pages 80–91. IEEE, Chicago, 1982.