# True Random Bit Generation

Anders Andersen[*] (20063516) Finn Jensen[†] (20060636)
Morten Kristensen[‡] (20063592)

December 17, 2009

[*] aa@cs.au.dk
[†] finnrj@cs.au.dk
[‡] netrom@cs.au.dk

# Contents

# 1 Introduction

Randomness is becoming a more and more important resource. Especially in cryptographic contexts where a crucial part of almost all cryptosystems is the ability to choose and generate large "random" quantities - typically primes or keys. This also implies that usage of inadequate sources of "randomness" will be a serious security flaw that can compromise even the strongest cryptographic protocol.

The main theme for this report will be a survey of possibilities for generating, processing, and testing random bits. Furthermore, we have conducted several practical experiments upon which we will draw our final conclusions and recommendations as to how you can "generate random bits" in different system settings and contexts.

## 1.1 What is Random?

In order to declare something "random" we will have to come up with a definition. As noted in [Knu69, 3.1] it does not make sense to talk of a number or a bit as being random. For instance, should the number 2 be declared random? Instead we will have to consider sequences of numbers or bits[1].

If we then pose certain restrictions on the distribution of the bit-sequences, we will be able to give a "statistical definition" of a random sequence. The restrictions are quite intuitive as the generation of a random bit-sequence can be thought of as the flipping of an unbiased "fair" coin. We would expect the result of each flip to be either "heads" or "tails" with `equal` probability and, furthermore, that the result of flipping the coin should be `independent` of any previous result. This leads to the following working definition which is identical to the definition in [Ken05, 3.1]:

**A random binary sequence:** *Let $X_n$ be a sequence of random variables, with $X_n \in \{0, 1\}$ for $n = 1, 2, 3 \ldots$ Then $X_n$ is a random sequence iff*

$$Pr(X_n = 1 \mid \textit{any distribution of all other } X_i, i \neq n) = 0.5$$

This definition is equivalent to the binary sequence being $\infty$-*distributed* which is defined using the notion of *k-distribution*:

**k-distributed binary sequence:** *A binary sequence is k-distributed iff*

$$Pr(X_n, X_{n+1} \ldots, X_{n+k-1} = x_1 x_2 \ldots x_k) = \tfrac{1}{2^k}$$

*for all binary sequences of length k.*

[1] We will in the rest of the report restrict the discussion to bit-sequences but the treatment of $b$-ary numbers is completely analogous.

**$\infty$-distributed binary sequence:** *A binary sequence is $\infty$-distributed iff it is k-distributed for all $k \in \mathcal{N}$*

**Remarks:** The above definition is primarily chosen for practical reasons as it makes it easier to understand most of the statistical tests that a random bit-sequence will be expected to pass (see section 5). The definition leads to the somewhat non-intuitive fact that a True Random Number Generator (TRNG) should generate a million length bit-string consisting of only 1s with the same probability as it generates any other more "random looking" bit-string of the same length.

A lot of definitions of "randomness" and "random sequences" have been proposed. A small survey are the following:

- Knuth goes through the motion of refining a definition based on $\infty$-distributed sequences in six steps before he is satisfied. Hereby adding the requirement of existence of effective algorithms determining infinite sub-sequences which only should be 1-distributed. Using induction this is enough to prove that they also will be $\infty$-distributed. After this Knuth continues by proving the existence of random sequences in order to ensure his definition is not too restrictive [Knu69, 3.5].

- Per Martin-Löf in [Martin-Löf, P. (1966). "The definition of random sequences". Information and Control 9: 602–619] defines randomness using Turing Machines: Given a finite b-ary sequence $x_1x_2, \ldots, n_N$ let $l(x_1x_2, \ldots, n_N)$ be the length of the shortest Turing Machine program generating this sequence. A random sequences of length $N$ is then defined as a sequence $s$ having $l(s) = Supremum\{l(s_i) \mid s_i$ has length N$\}$ [Knu69, 3.5]

- An interesting but alas not very practical definition is due to Mads Haahr, the owner and maintainer of random.org [Ran]:
  *"When it comes down to it, I think the most meaningful definition of randomness is that which cannot be predicted by humans."*

## 1.2 Applications of Random Numbers

Random numbers are essential components in many contexts. Apart from the use in cryptography a list of other applications is:

**Simulation:** In order to simulate complex systems, phenomena, or environments like e.g. particle systems in nuclear physics random numbers are required in order to simulate random behaviour.

**Gaming:** Internet gaming and lottery sites are becoming a fast growing group of consumers of random numbers which they use in order to simulate dice throws, card shuffling, draws etc.

**Sampling:** Random numbers can be used to select samples in cases where it is impractical or impossible to examine all possible cases.

**Programming:** Random numbers can be used as test-data for algorithms but are especially crucial to the design, analyzing and running of randomized algorithms.

**Decision-making:** Knuth mentions that many executives and even some college professors are rumored to decide their actions and grades by using coin flipping and the like. Random numbers can be used in all cases where a completely "unbiased" decision has to be taken [Knu69, 3.1].

**Art:** Artists are becoming more and more aware of the possibilities of using randomness to achieve aesthetic effects or to ensure uniqueness.

## 1.3 Types of Random Number Generators

Random number generators are divided into two groups depending on the way they generate random numbers. TRNGs are dependent on some source of natural occurring randomness and they generate bit-sequences by sampling this source.

Pseudo Random Number Generators (PRNGs) are based on deterministic algorithms exhibiting features that make their output statistically indistinguishable from output generated from TRNGs. They typically use the calculated output as input in order to create the next output. To bootstrap they therefore need an input (seed) which if unpredictability is essential itself has to be random. In cryptographic settings a PRNG thus requires a seed from a TRNG.

This report only concerns TRNGs but before we leave PRNGs a short comparison is appropriate. As mentioned in [RSN+08, 1.1.4], ironically, numbers generated by a PRNG often appear more random than sequences from a TRNG. A correctly implemented PRNG applies a number of transformations that eliminates statistical correlations between input and output. This means that PRNGs often have statistically better properties than TRNGs.

As observed in [Ken05, 3.3.3] TRNGs and PRNGs exhibit an interesting duality as advantages of one are disadvantages of the other. This can be illustrated in a table which thus also "by duality" characterize PRNGs [Ken05, table 3.1].

True Random Number Generators

| Advantages | Disadvantages |
|---|---|
| No periodicities | Slow and inefficient |
| No predictability based upon preceding sequences | Cumbersome to install and run |
| Certainty of non dependencies | Non reproducible sequences |
| High level of security | Expensive |
| Conceptually nice - not based on algorithms | Possible to manipulate |

## 2 Sources of Randomness

Historically published random tables have served as sources of randomness. In 1927 the statistician L.H.C. Tippett published a table consisting of 41.600 random numbers obtained by taking the middle digits from area measurements of English churches. In 1955 the Rand Corporation published the book: *A Million Random Numbers With 100.000 Normal Deviates.* The random source used can be compared to an electronic 32-place roulette wheel. [Sch96, 17.14] [McN08]

To qualify as a physical source of randomness a source should exhibit unpredictable and/or chaotic behaviour. Some sources used in practise are mentioned in the following.

**Radioactive Decay:** The decay of a radioactive isotope is considered to be unpredictable. Measurements of the time intervals between two successive decays can be used to construct bit-sequences. By comparing two successive intervals and e.g. defining the bit-value to be 1 in case the first interval is larger a random bit-sequence can be produced. This is the method used by Hotbits [Hot] where the decay of Kryton-85 into Rubidium is used. To avoid residual bias resulting from non-random systematic errors in apparatus or measure process the interpretation of 1s and 0s is interchanged for each bit.

**Atmospheric Noise:** This can be collected by tuning a radio to an unused frequency. The atmospheric noise picked up by the receiver can be sampled and used as the basis for a random pool. This is the method used by random.org [Ran] where the noise is sampled as an 8 bit mono signal at a frequency of 8 KHz. Of the 8 bits only the least significant is used. De-skewing is performed by using Transition Mappings as described in section 3.1.

**Thermal Noise:** Electronic devices such as semiconductor diodes or transistors can be used as sources of thermal noise.

A webcam with its lens cap on located in a closed "black box" can be used by measuring the thermal "noise" emitted by the webcam. This is the method used by LavaRnd [Lav]. The measurements are digitized and churned through an algorithm using n different parallel SHA-1 hash operations combined with n different xor-rotate and fold operations. This method produces considerable amounts of random bits reportedly of very high testing quality. Unfortunately the original and optically far more fascinating Lavarand site does not exist anymore. That site used webcam recordings of Lava Lites as its source.

**Oscillators:** Several chips using the frequency instability of a free running oscillators as the basis for random number generation have been constructed [Hrn]

**Turbulent flow:** In physics it is an unsolved problem, whether it is possible to make a theoretical model to describe the behavior of a turbulent flow - in particular, its internal structures. Using this fact one can use the air turbulence within a sealed disk drives, as a source of randomness. This turbulence causes fluctuations in rotating speed of the platters within the disk drive, and cause the access time to vary [DIF94].

In practise many applications draw upon more than one source of randomness and gathers the input from these into a pool. An example is `/dev/random` on the Linux platform which distills random bits from mouse and keyboard activity, disk Input/Output (IO) operations and specific interrupts.

# 3 Post-processing

Even if a sequence of should-be true random bits have been obtained (from a hardware source), it is still possible that it is biased and correlated. A biased sequence is a sequence where the number of ones and zeroes is not uniformly distributed. Or put in another way, if the probability of getting a one or a zero is not 1/2, then we have a biased sequence. Being correlated means that the bits depend, in some manner, on the value of the previous bits.

Eliminating bias and correlation is called `de-skewing` and a verity of different methods are known to solve these problems.

Even though de-skewing provides methods for transforming sequences of possibly biased and correlated bits into sequences of random bits it does not solve the problem of how best to gather a large amount of random bits. Enter `mixing`. As the name suggests, mixing techniques are $n$-ary ($n \geq 2$) and, therefore, need more than one input source. The general purpose is to mix the output of different uncorrelated sources together and, by doing so, increasing the randomness of the mixed pool.

## 3.1 De-skewing

### 3.1.1 Transition Mappings

Transition mappings are a cheap way of unbiasing sequences of data. One version proposed by von Neumann [ESC05, 4.2] is about examining bits in a sequence as non-overlapping pairs. All occurrences of 00 and 11 are discarded and all remaining occurrences of 10 will be replaced with 1 and 01 with 0. As an example: Assume we have a biased bit-sequence $X_n$ of length $N$. Let $Pr(X_n = 1) = 0.5 + E,\ n = 0, 1 \ldots N$ where $E$ denotes the *eccentricity* of the distribution: $0 < E \leq 0.5$. For bit-pairs we can make following table:

| Pair | Probability | | Generated bit |
|---|---|---|---|
| 00 | $(0.5-E)^2$ | $= 0.25 + E^2 - E$ | - |
| 01 | $(0.5-E)*(0.5+E)$ | $= 0.25 - E^2$ | 0 |
| 10 | $(0.5+E)*(0.5-E)$ | $= 0.25 - E^2$ | 1 |
| 11 | $(0.5+E)^2$ | $= 0.25 + E^2 + E$ | - |

As can be seen the probabilities for generating a 1 or a 0 are equal. Furthermore we have (denoting Expectation: $Exp$)

$$Pr(\text{generating a bit} \mid \text{a pair}) = 2*(0.25 - E^2)$$
$$\Downarrow$$
$$Exp(\text{number of pairs to generate one bit}) = \frac{1}{2*(0.25-E^2)}$$
$$\Downarrow$$
$$Exp(\text{number of bits to generate X bits}) = \frac{2*X}{2*(0.25-E^2)} = \frac{X}{0.25-E^2}$$

This technique will eliminate any bias but it has some restrictions. As can be seen above it will on average discard more than 75% of the bit-sequence. Moreover it is dependent on the bits coming from `identical independent distributions` and especially the bits must be uncorrelated for the above analysis to hold.

### 3.1.2 Fast Fourier Transform

Fast Fourier Transform (FFT) is used in many digital signal processing systems and in our case it is a most powerful algorithm for removing strong correlation and bias. This means that data which is strongly correlated but still exhibit some amount of entropy, can be extracted with the application of FFT [ESC05]. FFT is actually "just" an optimized variant of Discrete Fourier Transform (DFT), which runs in $O(n \log n)$ instead of $O(n^2)$, which obviously for large data set is warrented.

**DFT :** *Let $x_0, \ldots, x_{N-1}$ be a vector of complex numbers which is transformed to a sequence of $N$ complex numbers $X_0, \ldots, X_N$ by the formula:*

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k \frac{n}{N}} \qquad k = 0, \ldots, N-1$$

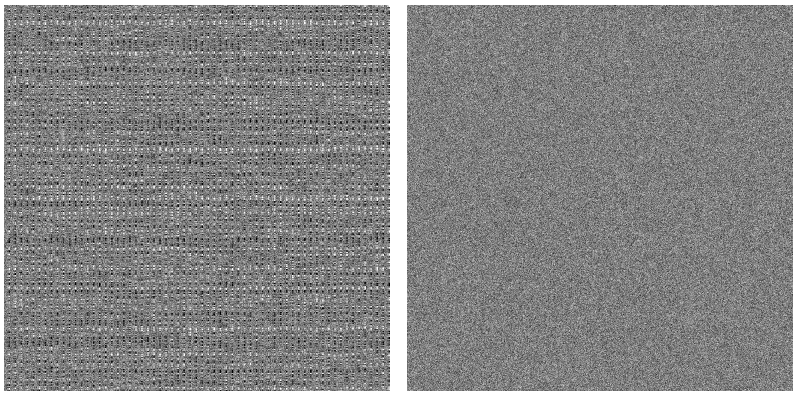
*where $i$ is the imaginary unit and $e^{\frac{2\pi i}{N}}$ is a primitive Nth root of unity, which means $(e^{\frac{2\pi i}{N}})^k \neq 1$ , $k = 0, \ldots, N-1$. According to [DIF94] the phase angle $\phi_k = \arg(X_k)$ is asymptotically independent and uniformly distributed on the interval $[0, 2\pi]$.*

In our application of FFT we "only" had real number measurements, which meant that for $N$ input we had $N/2 + 1$ outputs. We discarded the FFT predictable lines where the phase angles were equal to zero. Because $\phi_k$ is uniformly distributed on the interval $[0, 2\pi]$. To transform the random data we output 0 if the sign of the imaginary part was positive and 1 otherwise.

### 3.1.3 Compression

Compression is another way to de-skew, though crude it might be. It takes an input and produces a smaller output. The algorithm is reversible, though, and this is why no information is lost in the process. This is only possible, by Shannon information theory, if, on average, the shorter sequences are more uniformly distributed than the longer ones. Thus, the output is de-skewed [ESC05, 4.4]. For this technique to work in practice, the default headers added by the algorithms should be removed. Only the actual compressed data can be used.

To illustrate the need for Post-processing, a sample of unprocessed white noise has been used to generate the image shown in figure 1(a). The bad patterns are clearly seen, whereas after transformation the sample looks truly "noisy" figure 1(b).



(a) Unprocessed white noise.

(b) Fast Fourier Transformed white noise.

Figure 1: De-skewing illustrated.

## 3.2 Mixing

### 3.2.1 Bitwise Exclusive Or

Bitwise Exclusive Or (XOR) provides the means of a simple mixing function. It takes two sequences as input and utilizes the XOR function on every pair

of bits at the same index. XOR produces a 1 when the bits are equal and 0 otherwise. This also means that the resulting sequence will be as long as the input sequences, which have to be the same length. What has been obtained is that the result is less skewed, and if the inputs were not correlated then so much the better [ESC05, 5.1].

Compared to strong encryption algorithms and hashing functions, XOR is quite primitive and leaves much to be wanted.

### 3.2.2 Advanced Encryption Standard

Advanced Encryption Standard (AES) proves to be one of these strong mixing functions. It takes 128 bits of data and 256 bits as the key, which is 384 bits totally. From these it will output 128 bits as a result, which are de-skewed greatly as AES employs complex, and non-linear, functions. As suggested in [ESC05, 5.2], a powerful mixing scheme could be setup by utilizing, say, three input sources; A, B, and C. From these there exist a lot of ways of encrypting, for instance, A with B and the result with C. Each of them will provide some new, random, and de-skewed data.

### 3.2.3 Secure Hash Algorithm

Secure Hash Algorithm (SHA) (and family) also do a good job at mixing inputs. They map one sequence of input, which can be arbitrarily long, into some, possibly, smaller quantity. SHA-1 maps to 160 bits, SHA-256 to 256 bits, SHA-384 to 384 bits, and SHA-512 to 512 bits. The same trick, mentioned in the above paragraph, can be used here.

It is important to note that mixing functions, if they expand their output, *never* will expand the amount of randomness. If we have one random bit, we could e.g. XOR-mixing it with first a 0 and then a 1 getting output of two bits length. But as the output would be either 01 or 10, it would still only be worth one bit of randomness. [ESC05, 5.5].

# 4 Security Issues

A TRNG should adhere to several properties in order to be considered robust.

**Resilience:** The TRNG's output should look random to an adversary without knowledge of the internal state (randomness pool), even if the adversary has full control over data used to update the internal state.

**Forward Security:** Past output looks random to an adversary even if the adversary learns the internal state at a later time.

**Backwards Security/Break-in Recovery:** Future output looks random to an adversary with knowledge of the current internal state, provided the TRNG is updated with data having sufficient randomness.

The first formal description of a model to achieve this security was given in [BH05, 3] where it due to a broader definition was formulated in terms of `Robust Pseudo Random Number Generators`. It is though equally suited to our definition of TRNGs:

A `Robust TRNG` consists of two functions:

- $(r, s') \leftarrow \texttt{next}(s)$ where $s$ is the internal state (randomness pool). The function returns a $m$ length bit-sequence $r$ and replaces the internal state with the new state $s'$
- $s' \leftarrow \texttt{refresh}(s, x)$ where $x$ is a bit-sequence $|x| \geq m$ and $s$ the current internal state. The function updates the internal state to $s'$ using the data $x$.

A `Robust TRNG` should be able to play the role of the oracle in the following set-up: Let $A$ be a polynomial-time restricted adversary. Let $m$ be a security parameter (length of bit-sequence) and let $\mathcal{H}$ be a family of distributions over $\{0,1\}^{\geq m}$ i.e. the set of bit-sequences of size at least $m$. Let the adversary be equipped with four functions which he can call on the oracle: `good_refresh(.), bad_refresh(.), set_state(.)` and `next_bits()`.

**World 1 (The Real World):** The internal state of the oracle (TRNG) is initialised to null (i.e. $s \leftarrow 0^m$) and until the adversary decides to halt, he can interact with the oracle using the four functions with the following effect:

- $\texttt{good\_refresh}(\mathcal{D})$, $\mathcal{D} \in \mathcal{H}$ The oracle extracts a bit-sequences $x$ of length $m$ from $\mathcal{D}$ and sets the internal state to $s' \leftarrow \texttt{refresh}(s, x)$
- $\texttt{bad\_refresh}(x)$ with a bit-sequences $x$. The oracle set the internal state to $s' \leftarrow \texttt{refresh}(s, x)$
- $\texttt{set\_state}(s')$, $|s'| = m$. The oracle returns the current internal state $s$ to the adversary and sets the internal state to $s'$.
- `next_bits`() The oracle runs $(r, s') \leftarrow \texttt{next}(s)$ sets the internal state to $s'$ and returns $r$, $|r| = m$ to the adversary.

Upon halting the adversary outputs a bit $b$.

**World 0 (The Ideal World):** The ideal world is a bit more complicated (pardon the pun) as the oracle has to keep track of whether the adversary has compromised the TRNG by calling `set_state(.)` or not. For this purpose the oracle is equipped with a flag `compromised`.

The internal state of the oracle is initialised to null (i.e. $s \leftarrow 0^m$) and `compromised` $\leftarrow$ **true**. Until the adversary decides to halt, he can interact with the oracle using the four functions with the following effect:

- `good_refresh`($\mathcal{D}$), $\mathcal{D} \in \mathcal{H}$ The oracle resets `compromised` $\leftarrow$ **false**
- `bad_refresh`($x$) with a bit-sequences $x$. If `compromised` = **true** The oracle set the internal state to $s' \leftarrow$ `refresh`$(s, x)$. Otherwise it does nothing.
- `set_state`($s'$), $|s'| = m$. If `compromised` = **true** the oracle returns the current internal state $s$ to the adversary if this is not the case the oracle returns a random bit-sequences of length $m$ and returns this to the adversary. In either cases the oracle sets the internal state to $s'$ and `compromised` $\leftarrow$ **true**
- `next_bits`() If `compromised` = **true** the oracle runs $(r, s') \leftarrow$ `next`$(s)$, sets the internal state to $s'$ and returns $r$, $|r| = m$ to the adversary. If `compromised` = **false** the oracle returns a random bit-sequences of length $m$

Upon halting the adversary outputs a bit $b$.

Define $Pr_{A,i}(m, \mathcal{H})$ to be the probability that the adversary outputs 1 in world $i$. The advantage of $A$ is then defined to be:

$$\mathcal{A}dv_A(m, \mathcal{H}) = |Pr_{A,0}(m, \mathcal{H}) - Pr_{A,1}(m, \mathcal{H})|$$

**Robust TRNG:** *A TRNG is robust (with respect to a security parameter $m$ and distribution $\mathcal{H}$) if for any probabilistic polynomial-time adversary $A$ it holds that* $\mathcal{A}dv_A(m, \mathcal{H})$ is negligible in $m$.

In [BH05, 4] it is shown that this definition implies all the properties mentioned in the beginning of this section.

**Other Concerns:** There are some further problems which should be addressed when designing a TRNG.

**Protection against Denial of Service attacks:** The policy for usage of a TRNG should be considered. As an example the Linux `/dev/random` is implemented as a blocking process. Without any sort of user quota policy this gives an adversary the possibility of blocking the use of the TRNG. A simple command like: `dd if=/dev/random` is sufficient to block other users from reading `/dev/random` [GPR06].

**Guessable Passwords:** The usage of keyboard events should only include timings of keyboard pressing. If the pressed key-value is used directly a password enter by a user directly after booting the system might be an essential

part of feeding the initial randomness pool. An adversary with knowledge of the workings of the TRNG and the ability to consume enough random bits could possibly be able to determine the password by trying out all possible values and checking against the generated random bit-sequence.

**Protection of the Randomness Pool:** It is advisable to protect an adversary from easy access to the randomness pool. Assuming that an adversary is not in possession of administrator rights on the system (in which case we have probably lost anyway) the pool can be properly secured by having the strictest possible access permissions. This works fine as long as it also is possible to lock the randomness pool in memory in order to avoid it being paged to disk. That this is not always trivial is reported in [Gut98] where it is mentioned that the `Win32 VirtualLock()` function in Windows 95 was implemented as {`return TRUE;`}.

**Initial Randomness Pool at System Startup:** In order to be able to use the TRNG directly after booting the system enough randomness for the pool must be supplied. The most common solution is to hash or encrypt the old randomness pool at system shutdown and storing this value in a file. When rebooting the system this file can be used as an initial "seed" for the pool. Unfortunately, this leaves the problem of protection against an adversary either stealing the saved file from the disk or replacing it with another file of known content.

# 5 Testing for Randomness

When a sequence of bits has been produced, determining whether it is random is no straightforward procedure. In fact, it is impossible to mathematically prove anything to be random [MvOV01, 5.4]. Despite that, different strategies are known on how to detect the existence of certain wanted, and unwanted, properties.

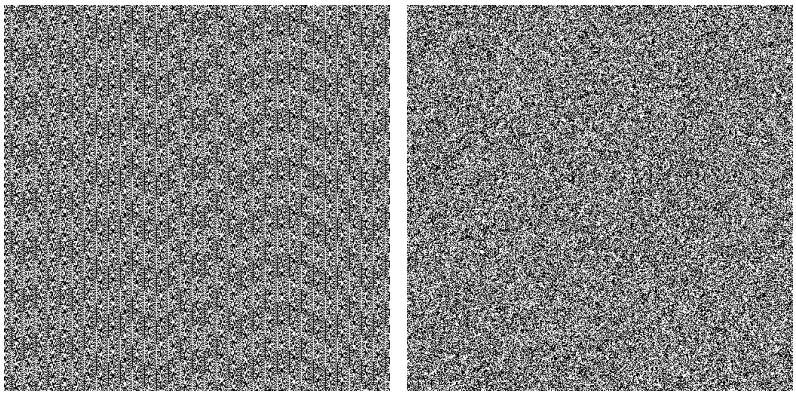
## 5.1 Optical Review

It is possible to check for randomness of a sequence by creating a picture of the bits and then asserting that there are no repetitive patterns in the picture. Scanning the raw bit-sequence, these repetitions would be "invisible" to the naked eye.

Figure 2 was generated using bits from a flawed version of the Windows PHP rand(0,1) function, whereas figure 2(b) was generated using data from random.org[2]. When these two are compared it is noticeable that figure 2(a) has patterns all over whereas figure 2(b) looks more like noise[3].

[2]Both figure 2(a) and 2(b) are borrowed from `http://www.random.org/analysis/`.

[3]This flaw was reported in April 2008. PHP does have a superior Mersenne Twister based mt_rand() function which does not show any such patterns.



(a) Windows version of PHP rand() function visualized.

(b) Data from random.org visualized.

Figure 2: Optical review.

Though this technique is not used much in practice, it serves as an interesting example for visualizing what randomness is.

## 5.2 Statistical Analysis

We will mainly use statistical analysis to attack the problem of determining randomness. Random number statistical tests determine the likelihood (or lack thereof) that a given finite set of data could have been produced by a true random source.

For a sequence of generated bits to pass a statistical test, the sequence must pass with the same level of confidence as an equivalent length sequence from a true random source. Additionally, if some sequences pass all tests in a given test suite, then instead of accepting them as random we think of them as not having been rejected – they have some characteristics of a random sequence but are not proven random.

More stringent we will apply `Statistical Hypothesis Testing` examining a bit-sequence $X_n$ with null-hypothesis, $\mathcal{H}_0$: *The bit-sequence is random* and the alternative hypothesis, $\mathcal{H}_a$: *The bit-sequence is not random.*

Each applied test will calculate a test statistic $Z(X_n)$. Using the probability distribution of the statistic a `p-value` can be calculated. This value, also referred to as `significance probability` or `observed significance level`, expresses the probability of observing a value for the test statistic that is at least as extreme as the value that was actually observed. The final decision on whether to reject the $\mathcal{H}_0$ hypothesis is made by comparing the *p-value* with the `significance level`, $\alpha \in (0, 1)$. That is $\mathcal{H}_0$ is rejected if *p-value* $\leq \alpha$ [BG03, 1.5].

It is, however, important to note that the unpredictability required in security-crucial environments is not the same as that provided with statistically tested randomness. Imagine throwing a die and encoding, in binary codes, the values as ASCII. The outcome of this is poor statistically speaking but has been produced from an unpredictable source. The opposite scenario could be to encrypt numbers from the consecutive enumeration $1, 2, 3, 4, 5, \ldots$ using AES with some predetermined key for all input. Here the input is predictable but the output is statistically sound [ESC05, pp. 6].

## 5.3 Selection of Test Suite

The problem of deciding which statistical tests to use is not trivial. Each test looks for one or more possible flaws in the output from the TRNG and since there are an infinite number of different statistical tests it is virtually impossible to address all aspects. As always with testing one also has to remember that tests can only prove presence never absolute absence of defects.

A different, worthwhile perspective is to contemplate which tests are good application-specific tests. Put in another way; which tests should be chosen to validate data for the specific application in question – data that could actually be used by the application. For instance, in a lottery it is not acceptable getting duplicates in the generated sequence. Dissimilar may be the requirements for rolling dice in various computer games.

The first proposal of a test suite for statistical testing of TRNGs was given by Knuth in [Knu69, 3.3]. Despite the advances in computer technology and Cryptology (which was not even mentioned by Knuth in 1969) this remained the "de facto" test suite for 25 years. So does the The Federal Information Processing Standards (FIPS) 140-1 test suite, which was issued in 1994, only mention 5 tests which all are part of Knuths test suite.

In 1995 George Marsaglia, Professor of Pure and Applied Mathematics and Computer Science and Statistics, published the Diehard test suite which introduced a number of more stringent tests that go beyond Knuth's, meanwhile classical but after today's measures, rather mild tests. In 1998 the ENT random number test suite was released by John Walker [Ent] and in 2001 The National Institute of Standards and Technology (NIST) implemented the Statistical Test Suite (STS) [RSN+08] as a combination of existing and newly developed tests.

On a selective basis, we have concluded that STS should be our main test suite mainly since it is the choice of the industry. Other important factors are that it has got the most stringent tests and it is designed to take binary input of arbitrary size. The FIPS-140-1 test suite was discarded on the basis of the latter case. The Diehard test-suite would also be interesting but it has not been maintained since the retirement of George Marsaglia.

## 5.4 Description of Chosen Tests

As we not always will be able to generate bit-sequence samples of the recommended minimum length for a given test, we will seldom be able to run all 15 tests in the test-suite. As a consequence we have chosen mainly to consider the 9 tests having the lowest minimum length requirements (see also section 6). Each of these is explained beneath [MvOV01, 5.4.4] [RSN+08].

### 5.4.1 Monobit Test

This test is also called the Frequency Test because it asseses whether the occurrence of ones and zeroes is equiprobable. It is defined by the following statistic

$$Z = \frac{(n_0 - n_1)^2}{n}$$

where $n_0, n_1$ denotes the number of zeroes and ones, respectively, and $n$ is the amount of bits in the input. It can be noted that $Z$ follows a $\chi^2$ distribution approximately with 1 degree of freedom when $n \geq 10$.

### 5.4.2 Cumulative Sums Test

The purpose of this test is to determine the maximal excursion away from Origo for a random walk [Wal] defined as cumulative sums of $s_n \in \{-1, 1\}$ where $s_n = 2 * X_n - 1$ for the original bit-sequence $X_n \in \{0, 1\}$. The test statistic calculated is

$$Z = max_{1 \leq k \leq n} |S_k|$$

### 5.4.3 Runs Test

A run of length $k$ is a series of either ones or zeroes that are separated by either the opposite type of bit or the start or end of the sequence. For instance, if we have the sequence 1001111010101, then there are four runs of ones of length 1, three runs of zeroes of length 1, one run of zeroes of length 2, and one run of ones of length 4. A `gap` of length $k$ is a run of $k$ zeroes and a `block` of length $k$ is a run of $k$ ones. Let $G_i$, $B_i$ denote the number of runs of length $i$ of gaps and blocks, respectively.

The test expects $e_i = (n - i + 3)/2^{i+2}$ number of gaps or blocks of length $i$ in the input sequence. Let $k$ be the largest integer $i$ for which $e_i \geq 5$. The statistic is defined as

$$Z = \sum_{i=1}^{k} \frac{(B_i - e_i)^2 + (G_i - e_i)^2}{e_i}$$

It can be noted that $Z$ follows a $\chi^2$ distribution approximately with $2k - 2$ degrees of freedom.

### 5.4.4 Serial Test

Here we check for occurrences of the two-bit pairs 00, 01, 10, and 11. As with the Monobit Test the number of occurrences should be approximately the same. The following statistic describes it

$$Z = \frac{4}{n-1}(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2) - \frac{2}{n}(n_0^2 + n_1^2) + 1$$

where $n_{00}$, $n_{01}$, $n_{10}$, and $n_{11}$ denote the number of occurrences of their respective two-bit pair. The occurrences are allowed to overlap. It can be noted that $Z$ follows a $\chi^2$ distribution approximately with 2 degrees of freedom when $n \geq 21$.

### 5.4.5 Approximate Entropy Test

As with the Serial Test the focus of the test is the frequency of all possible overlapping m-bit patterns across the bit-sequence. The purpose is here to compare the frequencies of two consecutive lengths (m and m+1) against the expected frequencies for a random sequence.

### 5.4.6 Discrete Fourier Transform Test

The DFT test also known as the spectral test checks for periodic, repetitive patterns of peak heights that are near each other. These periodic features indicate a deviation from the randomness assumed to be present. Concretely, it checks if the amount of peaks exceeding a 95% threshold deviates significantly from 5%

### 5.4.7 Poker Test

This is a generalization of the Monobit Test and is sometimes called the Block Frequency Test. Let $k = \lfloor \frac{n}{m} \rfloor$ where $m$ is a positive integer that satisfies $k \geq 5 \cdot (2^m)$ and $n$ the length of the bit-sequence. The input sequence is split into $k$ non-overlapping parts of length $m$. Let $n_i$ denote the number occurrences of the $i^{\text{th}}$ type of sequence, with $1 \geq i \geq 2^m$. These sub-sequences are all the possible outcomes (permutations) of a binary sequence of length $m$. The following statistic checks if the sub-sequences appears with expected frequency

$$Z = \frac{2^m}{k}\left(\sum_{i=1}^{2^m} n_i^2\right) - k$$

It can be noted that $Z$ follows a $\chi^2$ distribution approximately with $2^m - 1$ degrees of freedom.

### 5.4.8 Longest Run Test

The focus of this test is to determine whether the longest run of 1s with-in blocks of lengths M deviates significantly from the expected value for a random sequence. As any irregularity in the longest runs of 1s will also imply an irregularity in the longest run of 0s it is enough only to test only for 1s.

### 5.4.9 Binary Rank Test

The Binary Matrix Rank test checks for linear dependence between sub-sets of the sequence. Its focus is on the rank of disjoint sub-matrices of the sequence of data being tested. In practise the test uses 32 x 32 matrices constructed by taking consecutive blocks of length 32 from the bit-sequence as consecutive rows for the matrices. The observed number of ranks is then compared against the expected ranks for random matrices.

## 6 Experiments

The bit-sequences from each experiment were tested using the tests previously described in section 5.

**Test Setup:** We chose to use 100 streams whenever applicable i.e. each test was run 100 times, each time using a different sub-sequence. The length of each sub-sequence sampled from the binary input sequence was set according to the following table of minimum sizes (only consisting of the tests we used):

| Test | Min. input size |
|---|---|
| Frequency | 100 |
| Cumulative Sums | 100 |
| Runs | 100 |
| Serial | 500 |
| Approximate Entropy | 500 |
| DFT | 1024 |
| Block Frequency | 2000 |
| Longest Run | 6272 |
| Binary Matrix Rank | 38912 |

However, if the input was insufficient some of the tests were ignored. For instance, if we had 2MB of input then 2097152 bits/100 streams = 20971 < 38912, so the Binary Matrix Rank test was ignored.

Additionally, if the input length/100 streams < 100 (100 is the minimum length in the table above) then we only used a single stream. When this happened the outcome was less valuable but still provided some information.

**Remarks:** We chose to follow the recommendations in [RSN+08] literally although there might be some statistical concerns as to fixing an $\alpha$ value for all tests and to the independence of the different tests when running them on the same bit-sequence [Ken05, 4.2.1].

**Test Evaluation:** First of all, each experiment bit-sequence test was evaluated with a significance level $\alpha = 0.01$ as recommended in [RSN+08]. This means that for each test the bit-sequence tested was rejected as being random for a p-value below 0.01. Actually the results of this evaluation was not used directly. Instead an estimate of the proportion of passed tests in the streams was evaluated.

A TRNG will be expected infrequently to fail a test. In fact it is suspect if this should not be the case. The generated test report gives an estimate of the expected pass rate depending on the concrete number of streams (test runs). If a TRNG fails a type of test too often this is categorized as a `Proportional Failure`.

Another type of failure: `Uniformity Failure` is occurring if a TRNG fails in an inconsistent way. As the distribution of p-values for a test being run (say) 100 times should be uniformly distributed, it is possible to calculate a "p-value for the p-values". In practice this is done by dividing the $[0, 1]$ interval for the p-values into 10 buckets and counting occurrences in each bucket. This p-value is displayed directly in the output report, and should be $\geq 0.001$ due to the fact that the distribution of the p-value is $igamc\ (9/2,\ \chi^2/2)$ [4] A TRNG that has a few spectacular failures or experiences periods of poor statistical performance is likely to be declared vulnerable to uniformity failures.

A cryptographic strong TRNG must neither exhibit proportional nor uniformity failures.

**Remark:** In what follows we have chosen to use formulations like *"we can conclude that the bit-sequence was random"* if it passes all tests. We are aware that a statistical test cannot prove that a hypothesis is true, but a correct wording like *"on behalf of the tests the hypothesis cannot be rejected"* is awful cumbersome and somewhat non-intuitive.

## 6.1 Throwing a Die

**Setup:** In this experiment a die was thrown 512 times and encoded as a 1 if even and 0 if odd. This yielded 512 bits of data. As it is incredibly tedious and dull work, we could not produce thousands of bits as we would have liked for the statistical tests.

[4] $igamc$ is the incomplete gamma function used with 9 degrees of freedom as we have 10 buckets. And $\chi^2 = \sum_{i=0}^{10} \frac{(F_i - s/10)^2}{s/10}$, where $F_i$ denotes the number of items in the $i^{th}$ bucket and $s$ is the number of streams used.

**Test:** Given, that we only have 512 bits of data, the tests DFT, Block Frequency, Longest Run, and Binary Matrix Rank were ignored as they required more data. All other tests passed the analysis.

**Result:** Ultimately, throwing a die yields truly random data.
`http://www.daimi.au.dk/~netrom/crypto/test_reports/throwing_die` .

## 6.2 Random.org

**Setup:** Random.org [Ran] is an online service that supplies truly random bits of data. We downloaded a file from `http://www.random.org/files/` where a new file, containing one mebibyte or 8.388.608 bits, is added daily.

**Test:** It passed all tests.

**Result:** From the testing phase we conclude that Random.org supplies truly random data.
`http://www.daimi.au.dk/~netrom/crypto/test_reports/random.org` .

## 6.3 White Noise from a Microphone

Electrical noise is found in all electrical circuits and is characterised by being random. It also comes in different forms but we will consider white noise, or thermal noise as it is essentially the same. Thermal noise is generated by the fluctuations of the electric current inside an electrical conductor, caused by the random thermal motion of electrons.

Random noise can be obtained from a diversity of sources. Two of these are explained beneath.

**Audio input devices** provide noise. Recording from a microphone port, with no microphone hooked up, can be a source of white noise.

**Video input devices** such as a webcam can be used as well, but without the camera plugged in. The Lava RND project [Lav] takes another approach. They put a webcam in a dark box and record chaotic data. Here each pixel it produces consist mainly of digitized data measuring the background noise energy levels in a space completely devoid of light. Thus, it is a source of true randomness.

**Setup:** White noise was obtained by recording 2 minutes of data, which become 10.620.764 bytes, from a microphone port with no microphone attached to it. The Wave-formatted data was then transformed into 84.966.112 bits of ones and zeroes in textual representation.

**Test:** Initially, it failed all tests. Therefore, some de-skewing had to be employed with FFT, which yielded 42.483.056 bits as output. When sampled by STS it turned out to pass all tests.

**Result:** On the basis of the above, it was concluded that the obtained white noise was truly random. Surprisingly, this implied that the generator was able to produce 708.050 bits pr. sec. of highly random data.
`http://www.daimi.au.dk/~netrom/crypto/test_reports/white_noise_clean`
`http://www.daimi.au.dk/~netrom/crypto/test_reports/white_noise_fft` .

## 6.4 Mouse-pointer Device

**Setup:** A mouse-pointer device was monitored for a period of approximately one hour, as it was scrambled back and forth across the screen. Every move event captured produced a pair of coordinates, which were XOR'ed together. The procedure generated 732.427 bits.

**Test:** Because each stream could only consist of 7.324 bits the Binary Matrix Rank test was ignored. As expected, all tests failed. When FFT was applied every test, but DFT and Approximate Entropy, succeeded. This might be due to the fact that all 8 bits were used of each segment. Maybe the success-rate would have increased if only the least significant bit was used instead. Collecting the same amount of data would then have taken about 8 hours.

**Result:** Given that two crucial tests failed, the mouse data was deemed not "good enough" by itself.
`http://www.daimi.au.dk/~netrom/crypto/test_reports/mouse_clean`
`http://www.daimi.au.dk/~netrom/crypto/test_reports/mouse_fft` .

## 6.5 Network Traffic

**Setup:** To generate network traffic we pinged Google.com each 0.001 second for about 8 minutes and 20 seconds. Each successfully received reply was measured in milliseconds and only the decimal part was kept as resulting bits. 4.107.388 bits were generated.

**Test:** In the initial test phase, all tests failed as expected except for Binary Matrix Rank and FFT. We do not know why. After employing FFT all tests succeeded except for Approximate Entropy Test.

**Result:** Alas, the network traffic obtained is not a source for true randomness. As with the mouse-pointer data, it is not "good enough" by itself. From a cryptographic point of view, an adversary could be able to influence the traffic while gathering data. For instance, if Google simply replied in different patterns

of speed then they would affect our data.
`http://www.daimi.au.dk/~netrom/crypto/test_reports/ping_google_clean`
`http://www.daimi.au.dk/~netrom/crypto/test_reports/ping_google_fft`

## 6.6 Hard Drive

A hard drive basically consist of a enclosed box, with typically 6 to 12 platters and read/write heads for each platter but on the same actuator arm. These platters rotate with a speed of 5.400 RPM to 10.800 RPM, which creates turbulence at the read/write heads, between the disk surfaces and at the disk rims [DIF94] [Tan06].

The idea is that this turbulence is random and strong enough to influence the rotating speed of the disk. This means that the time it takes to read a single sector over an over again, would be influenced by the turbulence and give different access times for the same sector. These access times will be strongly biased and correlated, so some sort of de-skewing techniques is required - e.g. FFT.

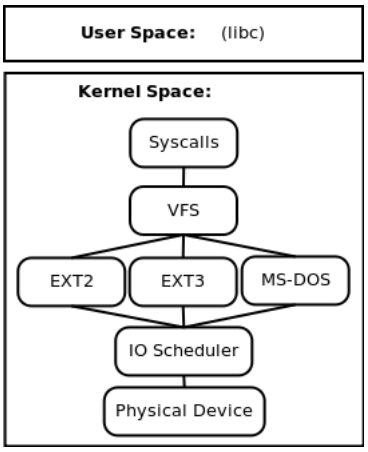



Figure 3: Linux Kernel illustrated.

**The Linux Kernel IO:** The Linux Kernel implements several abstractions to ease the use of IO, which implies many different ways of performing file read/write. The `User Space` open(), read(), and write() is the simplest and highest abstraction layer, which is the one to use when writing "normal" user applications. These command actually call the corresponding `Kernel Space` sys_open(), sys_read(), and sys_write(). These kernel calls are again an abstraction layer which call the corresponding Virtual File System (VFS) commands vfs_open(), vfs_read() and filp_open(). The VFS is yet again an abstractions layer, which distribute to the different file system implementations (fx EXT3, ms-dos...). These "file systems" send requests to the IO scheduler's "request

queue". The IO scheduler then send this request to the device driver, which handles the loading of sectors from the physical device [JC05].

**Implementation:** Several problems arose trying to implement this seemingly simple program. First off all the optimization done in the operating system, such as cache buffering, scheduler techniques and layers of abstractions yielded the biggest obstacle. This is all technologies that makes the "exact" timing of each sector read quite difficult.

Another issue to consider is how to exactly measure the different access times. High precision timers exist in the c language, which were obvious to use in User Space, but another method was needed in `Kernel Space`. Fortunately there are two ways to measure; one is to use "rdtsc" which count the clock cycles, the other is "jiffies" [5], which is used through out the kernel.

**User Space Program:** Our first implementation was a c-program that read a single block, from a hard drive in user space, with all options for buffering and "read ahead" disabled. The timing of the access for each block, typically yielded a big value, for the first block and then an significantly smaller value for the next. The first value is the only "real" value read from the physical hard drive, the rest were read from a buffer in memory. This is obviously not acceptable. The underlying reasons for this problem is that the program will be preempted by the scheduler and the cache buffering of files.

**Kernel Space - Module:** To circumvent the scheduler we implemented a Linux Kernel Module (LKM), in which we used the VFS abstraction to open a file from the kernel, and thereby making it possible to measure the access time. To avoid being preempted we simply disabled preemption for the time it took to do the measurements. Now we were sure that we did not get interrupted by other processes, but unfortunately the access times did not differ that much from the `User Space` program in that the first access time was a factor of three larger than the rest. This implies that the cache buffer was the biggest obstacle to overcome or perhaps the IO scheduler was taking too long time to finish the request.

**Kernel Space - IO Scheduler:** Diving further into the depths of the Linux kernel the next logical step was to look at the IO scheduler. The idea is to measuring the time it takes for each request to complete. This time is unfortunately *not* the access time for each sector, but the time it took to complete each request. By overwhelming the IO scheduler with request of a single sector, this should eliminate this problem. Some possible unwanted timings is done, e.g. the operating system writes to a log file, which also would be timed.

[5] Within the Linux 2.6 operating system kernel, since release 2.6.13, on the Intel i386 platform a jiffy is by default 4 ms, or $\frac{1}{250}$th of a second. The jiffy values for other Linux versions and platforms have typically varied between about 1 ms and 10 ms.

**Setup:** We used a Linux kernel 2.6.32 and changed the IO-scheduler to record each time a request finished and reported this result in a convenient way. We created three test-cases: (1) reading continuously 1 block from the harddrive, (2) normal usage of the OS, (3) reading continuously 10GB of data to /dev/null. Obviously the amount of data differed considerably. After the data had been collected, the variations in the measurements were run through the FFT and tested with the test suite. (1) collected 978.569 bits of data in 4 hours, (2) 34.619 bits in 12 hours, (3) 1.216.283 bits in 12 hours.

**Test:** (1) Unfortunately, all the test failed except Binary Matrix Rank, which was ignored because the input was less that 38.912 pr. stream. (2) because of the low amount of data we only ran the test suite with one stream. It passed the tests; Frequency, Block Frequency, and Cumulative Sums. (3) All test cases failed.

**Result:** The failing test cases implies that the collected bits are not to be used in Cryptography. There are several ways in which something could have gone wrong. First of all, the "original" experiment in [DIF94] used a 40Mb 5.25 inch hard disk, while the test machine had a 60 Gb 2.5 inch hard disk. So one could make the hypothesis that because the amount of air is much less, the air turbulence generated within it is to weak. The Linux kernel is a huge "monster", so misunderstanding the kernel architecture is not unlikely. Lastly delays in the IO-controller or bus arhitecture could influence the timing of each block.
(1) `http://www.daimi.au.dk/~netrom/crypto/test_reports/hdd1B_resFFTReport`
(2) `http://www.daimi.au.dk/~netrom/crypto/test_reports/hddNormal_resFFTReport`
(3) `http://www.daimi.au.dk/~netrom/crypto/test_reports/hdd10GB_resFFTReport`

## 6.7 /dev/random

The Linux `/dev/random` is the most popular open source implementation of a TRNG. It is embedded in all Linux environments. The output is used internally by the kernel itself and is also available to users via two device drivers named `/dev/random` and `/dev/urandom` respectively.

The difference lies in the stated quality of the random bits and the delay for receiving them. By the wording of the Linux man page [man]: *"/dev/random should be suitable for uses that need very high quality randomness such as one-time pad or key generation."* The driver is blocking in case the randomness pool is empty. This is not the case for `/dev/urandom` but as the man page mention: *"As a result, if there is not sufficient entropy in the entropy pool, the returned values are theoretically vulnerable to a cryptographic attack on the algorithms used by the driver."* It should be noted that these security claims have been severely criticised [BH05, 5.3] and that an attack which breaks the `forward-security` on the kernel version 2.6.10 was shown in [GPR06].

The architecture of the Linux TRNG consists of three asynchronous processes. The first process is responsible for the gathering of random bits from various kernel events (mouse, keyboard, disk and interrupts). The second process collects these bits and uses a mixing function to build up and fill a randomness pool (The Primary Entropy Pool). This pool feeds two secondary pools (Secondary Entropy Pool and Urandom Entropy Pool) and the feeding of these pools together with pool updating and output generation is the responsibility of the third process. The only non-linear cryptographic operation used by all processes is the SHA-1 function.

**Setup:** 19.580.072 bits were extracted from the device `/dev/random` in a period of approximately 28 hours. The version of the kernel was 2.6.32. While extracting data we ran other "entropy gathering" programs in the background such as `dd`, `find`, and `ls`. This was due to the fact that `/dev/random` blocks whenever its amount of entropy drops to zero.

**Test:** It passed all tests.

**Result:** As was expected, `/dev/random` provides a source of truly random bits of data.
`http://www.daimi.au.dk/~netrom/crypto/test_reports/dev_random`

# 7 Conclusion

The objective of this report has been the mission of defining, gathering, testing, and experimenting with truly random data.

First off, we defined a random sequence statistically as a $\infty$-distributed binary sequence. Second, several sources were found which qualified for true randomness by exhibiting unpredictable and/or chaotic behaviour.

In the post-processing phase we first discussed different techniques for deskewing data and thereby eliminating bias and correlation factors. One thing that came as a surprise was the fact that a lot of data was thrown away when these techniques were used. For instance, FFT throws at least 50% away and von Neumann throws at least 75% away. This made the operation of gathering data even more cumbersome and time-consuming.

Since it is mathematically impossible to prove anything to be random, we had to come up with something else and we chose the approach of statistical analysis. Amongst FIPS-140-1, ENT, Diehard, and STS, the last test suite was our choice. Simply because it is used in the industry and has very stringent tests.

Ultimately, we concluded that the best source, amongst those we tried, was thermal noise recorded from a microphone port with no microphone attached. Not only was it superior in the test results but the speed in which we could

obtain data was 708.050 bits/second. `/dev/random` was also a very good choice as it resides in almost every *nix distribution. If a generic program was required and no soundcard was available then the choice would be `/dev/random` indeed. Unfortunately, mouse-pointer data, network traffic (the pinging of Google), and the Hard Drive tests failed.

# References


[BG03] Preben Blæsild and Jørgen Granfeldt. *Statistics with Applications in Biology and Geology*. Chapman & Hall London, CRC, Boca Raton, Florida., 2003.

[BH05] Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. *Proceedings of the 12th ACM conference on Computer and communications security*, 2005. http://www.cs.princeton.edu/~boaz/Papers/devrand.pdf.

[DIF94] D Davis, R Ihaka, and P Fenstermacher. Cryptographic randomness from air turbulence in disk drives. *Advances in Cryptology - Crypto '94*, 1994. http://world.std.com/~dtd/random/forward.pdf.

[Ent] Ent. http://www.fourmilab.ch/random/.

[ESC05] D Eastlake, J Schiller, and S Crocker. Rfc 4086. randomness requirements for security. *The Internet Society*, 2005. http://www.ietf.org/rfc/rfc4086.txt.

[GPR06] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006. http://eprint.iacr.org/2006/086.pdf.

[Gut98] Peter Gutmann. Software generation of practically strong random numbers. *Proceedings of the 7th USENIX Security Symposium*, 1998. http://www.usenix.org/publications/library/proceedings/sec98/full_papers/gutmann/gutmann.pdf.

[Hot] Lavarnd. http://www.fourmilab.ch/hotbits/.

[Hrn] Hardware random number generator. http://en.wikipedia.org/wiki/Hardware_random_number_generator.

[JC05] Alessandro Rubini Jonathan Corbet, Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly February 2005, 2005.

[Ken05] C Kenny. Random number generators: An evaluation and comparison of random.org and some commonly used generators. *The Distributed Systems Group*, 2005. http://www.random.org/analysis/Analysis2005.pdf.

[Knu69] Donald E. Knuth. *The art of computer programming, volume 2 : seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969. http://portal.acm.org/citation.cfm?id=270146.

[Lav] Lavarnd. http://www.lavarnd.org/.

[man] */dev/random /dev/urandom*. www.kernel.org/doc/man-pages/online/pages/man4/random.4.html.

[McN08] Tom McNichol. Totally random. *Wired 11, 2008*, 2008. http://www.wired.com/wired/archive/11.08/random.html.

[MvOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001. http://www.cacr.math.uwaterloo.ca/hac/.

[Ran] Random.org. http://www.random.org/.

[RSN+08] A Rukhin, J Soto, J Nechvatal, M Smid, E Barker, S Leigh, M Levenson, M Vangel, D Banks, A Heckert, J Dray, and S Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. *NIST*, 2008. http://csrc.nist.gov/publications/nistpubs/800-22-rev1/SP800-22rev1.pdf.

[Sch96] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. Wiley & Sons, 1996.

[Tan06] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall India; 5th edition (2005), 2006.

[Wal] Random walk. http://en.wikipedia.org/wiki/Random_walk.

# Acronyms

| | |
|---|---|
| **AES** | Advanced Encryption Standard |
| **ASCII** | American Standard Code for Information Interchange |
| **DFT** | Discrete Fourier Transform |
| **FFT** | Fast Fourier Transform |
| **FIPS** | The Federal Information Processing Standards |
| **IO** | Input/Output |
| **LKM** | Linux Kernel Module |
| **NIST** | The National Institute of Standards and Technology |
| **PHP** | PHP: Hypertext Preprocessor |
| **PRNG** | Pseudo Random Number Generator |
| **SHA** | Secure Hash Algorithm |
| **STS** | Statistical Test Suite |
| **TRNG** | True Random Number Generator |
| **VFS** | Virtual File System |
| **XOR** | Bitwise Exclusive Or |