The maxwell(8) random number generator (v2)

Sandy Harris
sandyinchina@gmail.com

Abstract:

I propose a demon process for use on Linux. It gathers entropy from timer calls, distills in into a concentrated form, and sends it to the kernel random(4) device. The program is small and does not require large resources. The entropy output is of high quality. The output rate is a few kilobits per second, which is enough for nearly all applications.

This paper describes version 2 of maxwell (2016), incorporating various changes suggested by others plus a few of my own. The main change from v.1 (2012) is that the command line options are different, making the user interface simpler.

Overview

Random numbers are essential for most cryptographic applications, and several otherwise quite good cryptographic systems have been broken because they used inadequate random number generators. The standard reference is RFC 4086, Randomness Requirements for Security [1]. It includes the following text:

> At the heart of all cryptographic systems is the generation of secret, unguessable (i.e., random) numbers. The lack of generally available facilities for generating such random numbers (that is, the lack of general availability of truly unpredictable sources) forms an open wound in the design of cryptographic software. [1]

However, generating good random numbers is often problematic. The same RFC also says:

> Choosing random quantities to foil a resourceful and motivated adversary is surprisingly difficult. This document points out many pitfalls … [1]

I will not belabour these points here. I simply take it as given both that high-quality random numbers are important and that generating them can be rather a tricky proposition.

The Linux random device

Linux provides a random number generator in the kernel; it works by gathering entropy from kernel events, storing it in a pool, and hashing the pool to produce output. It acts as a device driver supporting two devices:

- /dev/random provides high-grade randomness for critical applications and will block (make the user wait) if the pool lacks entropy
- /dev/urandom never blocks (always gives output) but is only cryptographically strong, and does not give guaranteed entropy

The main documentation is the manual page, random(4); the source code also has extensive comments. Archives of the Linux kernel mailing list and other lists have much discussion. A critique [16] of an earlier version has been published.

In many situations, the kernel generator works just fine with no additional inputs. For example, a typical laptop or desktop system does not do a great deal of crypto, so the demands on the generator are not heavy. On the other hand, there are plenty of inputs – at least keyboard and mouse activity plus disk interrupts.

On other systems, however, the kernel generator may be starved for entropy. Consider a Kerberos server which hands out many tickets, or a system with many encrypted connections, whether IPsec, SSL/TLS or SSH. It will need considerable randomness, but such servers often run headless – no keyboard or mouse – and entropy from disk events may be low. There may be a good deal of network activity, but some of that may be monitored by an enemy, so it is not a completely trustworthy entropy source.

If the kernel generator runs low on entropy, then a program attempting to read /dev/random will block; the device driver will not respond until it has enough entropy so the user program must be made to wait. A program reading /dev/urandom will not block but it cannot be certain it is getting all the entropy it expects. The driver is cryptographically strong and the state is enormous, so there is good reason to think the outputs will be of high quality, but there is no longer a guarantee. Whichever device they read, programs and users relying on the kernel generator may encounter difficulties if the entropy runs low. Ideally, that would never happen.

Problem statement

The kernel generator provides an interface that allows an external program to provide it with additional entropy, to prevent any potential shortage. The problem we want to solve here is to provide an appropriate program. The entropy volume need not be large, but the quality should be high.

An essential requirement is that the program not overestimate the entropy it is feeding in, because sufficiently large mis-estimates repeated often enough could cause the kernel generator to misbehave. This would not be easy to do; that generator has a huge state and is quite resilient against small errors of this type. However, frequent and substantial errors could compromise it.

Underestimating entropy is much less dangerous than overestimating it. A low estimate will waste resources, reducing program efficiency. However, it cannot compromise security.

I have written a demon program which I believe solves this problem. I wanted a name distinct from the existing “Timer entropy demon” [2], developed by Folkert van Heusden, so I named mine maxwell(8), after Maxwell's demon, an imaginary creature discussed by the great physicist James Clerk Maxwell. Unlike its namesake, however. my program does not create exceptions to the laws of thermodynamics.

Existing generators

There are several good ways to get randomness to feed into the kernel generator already. In many – probably even most – cases, one of these will be the best choice and my program will not be necessary. Each of them, however, has disadvantages as well, so I believe there is still a niche which a new program can fill.

Ideally, the system comes with a built-in hardware RNG and failing that, there are other good alternatives. I limit my discussion to three – Turbid, HAVEGE and Cryptlib – each of which has both Open Source code and a detailed design discussion document available. As I see it, those are minimum requirements for a system to inspire confidence.

Also, the authors of all those generators are (or have been) affiliated with respectable research institutions and have PhDs and publications; this may not be an essential prerequisite for trusting their work, but it is definitely reassuring.

Built-in hardware

Where it is available, an excellent solution is to use a hardware RNG built into your system. Intel have one in some of their chipsets, Via build one into some CPU models, and so on. If one is buying a server that will do a significant amount of crypto, then insisting on a hardware RNG as a requirement in the specification is completely reasonable. In today’s market, this does not much restrict your choices.

The main difficulty of with this method is that not all systems are equipped with these devices. You may not get to choose or specify the system you work on, so the one you have may lack a hardware RNG even if your applications really need one. Even if the device is present, there will not necessarily be a Linux driver available. In some cases, there might be deficiencies in the documentation required to write a driver, or in the design disclosure and analysis required before the device can be fully trusted. In short, this is usually the best choice when available, but it is not universally available.

A true paranoid might worry about an intelligence agency secretly subverting such a device during the design or manufacturing process. Some of Edward Snowden’s disclosures can be read as indicating that the NSA already do this, though other interpretations are possible. Given that a chip might be designed in the Israeli office of an American company and manufactured in a Chinese factory controlled by a Taiwanese company, there are many agencies one could worry about.

However, I do not think this is a particularly realistic worry. For one thing, intelligence agencies no doubt have easier and more profitable targets to go after. Also, if the hardware RNG feeds into random(4) then – as long as there is some other entropy – the large driver state plus the complex mixing would make it extremely difficult to compromise that driver even with many of its inputs known. This means that – while directly using hardware RNG output may be risky and the exact level of risk is extremely hard to assess – using it to feed the Linux device is almost certainly safe.

Adding a good second source of entropy – maxwell(8) or any of those discussed below – makes an attack via RNG subversion utterly implausible.

Turbid

John Denker's Turbid – a demon for extracting entropy from a sound card or equivalent device, with no microphone attached – is another excellent choice. It can give thousands of output bytes per second, enough for almost any requirement.

Turbid is quite widely applicable; many motherboards include a sound device and on a server, this is typically unused. Failing that, it may be possible to add a device, either internally if the machine has a free slot or externally via a USB port. Turbid can also be used on a system which uses its sound card for sound. Add a second sound device; there are command-line options which will tell Turbid to use that, leaving the other card free for music, VoIP or whatever.

The unique advantage of Turbid is that it provably delivers almost perfectly random numbers. Most other generators – including mine, random(4), and the others discussed in this section – estimate the randomness of their inputs. Sensible ones attempt to measure the entropy, and are very careful that their estimates are sufficiently conservative. They then demonstrate that, *provided that the estimate is good*, the output will be adequately random. This is a reasonable approach, but hardly optimal.

Turbid does something quite different. It measures physical properties of the sound device and uses arguments from physics to derive a lower bound on the Johnson-Nyquist noise [3] which must exist in the circuit. From that, and some mild assumptions about properties of the hash used, it gets a *provable lower bound on the output entropy*. Parameters are chosen to make that bound 159.something bits per 160-bit SHA context. The documentation talks of “smashing it up against the asymptote”.

However, Turbid also has disadvantages. It requires a sound card or equivalent, a condition that is easily satisfied on many systems but may be impossible on some. Also, if the sound device is not already known to Turbid, then a measurement step is required before program parameters can be correctly set. These are analog measurements, something some users may find inconvenient.

The Turbid web page [4] has links to the code and a detailed analysis.

HAVEGE

The HAVEGE (HArdware Volatile Entropy Gathering and Expansion) RNG gathers entropy from the internal state of a modern superscalar processor. There is a demon for Linux, haveged(8), which feeds into random(4).

The great advantages of HAVEGE are that the output rate can be very high, up to hundreds of megabits a second, and that it requires no extra hardware – just the CPU itself. For applications which need such a rate, it may be the only solution unless the system has a fast built-in hardware RNG.

However, HAVEGE is not purely a randomness gatherer:

> HAVEGE combines entropy/uncertainty gathering from the architecturally invisible states of a modern superscalar microprocessor with a pseudo-random number generation [5]

The “and Expansion” part of its name refers to a pseudo-random generator. Arguably, this makes

HAVEGE less than ideal as source of entropy for pumping into random(4) because any pseudo-random generator falls short of true randomness, by definition. In this view one should either discard the “and Expansion” parts of HAVEGE and use only the entropy gathering parts, or use the whole thing but give less than 100% entropy credit.

There is a plausible argument on the other side. Papers such as Yarrow [7] argue that a well-designed, well-implemented and well-seeded PRNG can give output good enough for cryptographic purposes. If the PRNG output is effectively indistinguishable from random, then it is safe to treat it as random. The HAVEGE generator appears to meet this criterion; it depends on internal processor state which is not visible to users, hence not knowable by an opponent, and the generator’s state is continuously updated. The haveged(8) demon therefore gives full entropy credit for HAVEGE output.

Another difficulty is that HAVEGE seems to be extremely hardware-specific. It requires a superscalar processor and relies on:

> a large number of hardware mechanisms that aim to improve performance: caches, branch predictors, … The state of these components is not architectural (i.e., the result of an ordinary application does not depend on it). [6]

This will not work on a processor that is not superscalar, nor on one to which HAVEGE has not yet been ported.

Porting HAVEGE to a new CPU looks difficult; it depends critically on “non-architectural” features. These are exactly the features most likely to be undocumented because programmers generally need only a reference to the architectural features, the ones that can affect “the result of an ordinary application”. These “non-architectural” aspects of a design are by definition exactly the ones which an engineer is free to change to get more speed or lower power consumption, or to save some transistors. Hence, they are the ones most likely to be different if several manufacturers make chips for the same architecture, for example Intel, AMD and Via all building x86 chips or the many companies making ARM-based chips. They may even change from model to model within a single manufacturer's line; for example Intel's low power Atom is different internally from other Intel CPUs.

On the other hand, HAVEGE does run on a number of different CPUs, so perhaps porting it actually simpler than it looks. It seems possible that one need not care exactly which non-architectural features a given chip has; as long are there are some such features, timings will vary and HAVEGE will work.

HAVEGE, then, appears to be a fine solution on some CPUs, but it may be no solution at all on others.

The HAVEGE web page [6] has links to both code and several academic papers on the system. The haveged(8) web page [15] has both rationale and code for that implementation.

Cryptlib

Peter Gutmann's Cryptlib includes a software RNG which gathers entropy by running Unix commands and hashing their outputs. The commands are things like ps(1) which, on a reasonably busy system, give changing output. The great advantage is that this is a pure software solution. It should run on more-or-less any system, and has been tested on many. It needs no special hardware.

One possible problem is that the Cryptlib RNG is a large complex program, perhaps inappropriate for some systems. On the version I have (3.4.1), the random directory has just over 50,000 lines of code (.c .h and .s) in it, though much of that code is machine-specific and the core of the RNG is no doubt far smaller. Also the RNG program invokes many other processes so overall complexity and overheads may be problematic on some systems.

Also, the RNG relies on the changing state of a multi-user multi-process system. It is not clear how well it will work on a dedicated system which may have no active users and very few processes.

The Cryptlib website [8] has the code and one of Gutmann's papers [9] has a detailed rationale.

Our niche

Each of the alternatives listed above is a fine choice in many cases. Between them they provide quite a broad range of options. What is left for us?

What we want to produce is a program with none of the limitations listed above. It should not impose any hardware requirements, such as:

- requiring an on-board or external hardware RNG
- requiring a sound card or equivalent device like Turbid
- requiring certain CPUs as HAVEGE seems to

Nor should it be a large complex program, or invoke other processes, as the Cryptlib RNG does.

Our goal is the smallest simplest program that gives good entropy. I do at least get close to this; the compiled program is small, resource usage is low, and output quality tests as high.

Choice of generator

In the most conservative view, only a generator whose inputs are from some inherently random process such as radioactive decay or Johnson-Nyquist circuit noise should be fully trusted – either an on-board hardware RNG or Turbid. In this view other generators – random(4), maxwell(8), HAVEGE, Cryptlib, Yarrow, Fortuna, … – are all in effect using system state as a pseudo-random generator, so they cannot be trusted completely.

Taking a broader view, any well-designed generator – one whose output is indistinguishable from random even for an opponent with large resources – can be used, so all those discussed here are usable in some cases; the problem is to choose among them.

If there is a hardware RNG on your board or you have a sound device free for Turbid, then that is the clearly the generator to use. Either of those uses inputs that are provably inherently random, and can give large amounts of high grade entropy for little resource cost. If both are available and you have high security requirements, use both.

If no hardware RNG is available, the choice is more difficult. It is possible to use maxwell(8) in all cases, but using the Cryptlib RNG or HAVEGE is practical on many systems and may be preferable on some. Adding a sound device for Turbid or using an external hardware RNG is worth considering as well, especially if the system will do a lot of crypto and have heavy demand for random numbers.

Applications for maxwell(8)

There are several situations where maxwell(8) can be used:

- where the generators listed above are, for one reason or another, not usable
- when using one of the above generators would be expensive or inconvenient.
- as a second generator run in parallel with any of the above, for safety if the other fails
- when another generator is not fully trusted (“Have the NSA got to Intel?” asks the paranoid)
- whenever a few K bits a second is clearly enough
- when a few extra K bits are needed

There are many applications:

- maxwell(8) can be used even on a very limited system – an embedded controller, a router, a plug computer, a Linux cell phone, ... Some of these may not have a hardware RNG, or a sound device that can be used for Turbid, or a CPU that supports HAVEGE. The Cryptlib RNG is not an attractive choice for a system with limited resources and perhaps a cut-down version of Linux that lacks many of the programs that the RNG program calls. In such cases, maxwell(8) may be the only reasonable solution.
- Using any generator alone gives a system with a single point of failure. Using two is a sensible safety precaution in most cases, and maxwell(8) is cheap enough to be quite suitable as the second, whatever is used as the first.
- maxwell(8) runs faster at startup to fill up the entropy pool, then switches to more conservative parameters for long-term use. The *-f* option controls how much of the initial **f**ast output is done. Without the option, it does 4 K bits of fast output, then switches parameters.
- Before some randomness-intensive action such as generating a large PGP key, one can run maxwell(8) to fill up the entropy pool. The *-h* option makes the program **h**alt after a fixed amount of output; *-f* can be used as well.
- Some machines may have heavy demands at certain times, for example at the start of the work day. Typically, just having maxwell running in the background will be enough to cope with this, but one might also have a cron job load up the pool just before the anticipated demand.

maxwell(8) can be used along with other generators. The computer I am writing this on has Intel’s hardware RNG but uses maxwell as a second entropy source. This is overkill for a laptop; almost certainly either would be enough, and it might even be fine with neither. However, something like that might be exactly what is needed on a busy server.

Design overview

The old joke "Good, fast, cheap – pick any two." applies here, with:

- good == excellent randomness
- fast == high volume output
- cheap == a small simple program

I choose good and cheap. We want excellent randomness from a small simple program; I argue both that this is possible and that my program actually achieves it.

Choosing good and cheap implies not fast. Some of the methods mentioned above are extremely fast; we cannot hope to compete, and do not try.

Randomness requirements

Extremely large amounts of random material are rarely necessary. The RFC has:

> How much unpredictability is needed? Is it possible to quantify the requirement in terms of, say, number of random bits per second? The answer is that not very much is needed. ... even the highest security system is unlikely to require strong keying material of much over 200 bits. If a series of keys is needed, they can be generated from a strong random seed (starting value) using a cryptographically strong sequence ... A few hundred random bits generated at start-up or once a day is enough if such techniques are used. … [1]

There are particular cases where a large burst is needed; for example, to generate a PGP key, one needs a few K bits of top-grade randomness. It might also be useful to have a cron job dump some data into the pool periodically, at least "A few hundred random bits ... once a day", but if maxwell(8) is run in the background it continuously tops up the pool so that may not be necessary.

In general even a system doing considerable crypto will not need more than a few hundred bits per second of new entropy. For example, if an IPsec gateway supports 300 connections and rekeys each of them every 20 minutes, then it will do 900 rekeys an hour, one every four seconds on average. Even if each rekey needed 2048 bits and for some reason it needed the quality of /dev/random, the kernel would need only 512 bits of input entropy per second to keep up. maxwell(8) is about an order of magnitude faster than that, giving a few K bits per second; details are in the "Resources and speed" section.

In a more realistic scenario – session keys are only a few hundred bits and the quality of /dev/urandom is enough for them – maxwell(8) is at least two orders of magnitude faster than the requirement.

Of course a system that does a great deal of crypto – the IPsec gateway in the example above, or a server doing a lot of Kerberos, SSH or TLS – really should have a hardware RNG (choose a machine with an on-board RNG or use Turbid) but it look as though maxwell(8) would be adequate on at least some such systems. The RFC suggests that for many systems “A few hundred random bits generated at start-up or once a day is enough”, and maxwell(8) is surely more than enough for those.

Timer entropy

The paper “Analysis of inherent randomness of the Linux kernel” [10] includes tests of how much randomness one gets from various simple sequences. The key result for our purposes is that (even with interrupts disabled) just:

- doing usleep(100), which gives a 100 µs delay
- doing a timer call
- taking the low bit of timer data

gives over 7.5 bits of measured entropy per output byte, nearly one bit per sample.

Both the inherent randomness [10] and the HAVEGE [5] papers also discuss sequences of the type:

- timer call
- some simple arithmetic
- timer call
- take the difference of the two timer values

They show that there is also entropy in these. The time for even a simple set of operations can vary depending on things like cache and TLB misses, interrupts, and so on.

There appears to be enough entropy in these simple sequences – either usleep() calls or arithmetic – to drive a reasonable generator. That is the basic idea behind maxwell(8). The sequence used in maxwell(8) interleaves usleep() calls with arithmetic, so it gets entropy from both timer jitter and differences in time for arithmetic.

On the other hand, considerable caution is required here. The RFC has:

> Computer clocks and similar operating system or hardware values, provide significantly fewer real bits of unpredictability than might appear from their specifications. Tests have been done on clocks on numerous systems, and it was found that their behavior can vary widely and in unexpected ways. ... [1]

My design is conservative. For each 32-bit output, it uses at least 48 clock samples, so if there is 2/3 of a bit of entropy per sample then the output has 32 bits. Then it tells random(4) there are 30 bits of entropy per output delivered. If that is not considered safe enough, the -p (paranoia) command-line option allows the administrator to increase the number of samples per output.

maxwell(8) uses a modulo operation rather than masking to extract bits from the timer, so more than one bit per sample is possible. This technique also helps with some of the possible oddities in clocks which the RFC points out:

> One version of an operating system running on one set of hardware may actually provide, say, microsecond resolution in a clock, while a different configuration of the "same" system may always provide the same lower bits and only count in the upper bits at much lower resolution. This means that successive reads of the clock may produce identical values even if enough time has passed that the value "should" change based on the nominal clock resolution. [1]

Taking only the low bits from such a clock is quite problematic; the result may never change. However, extracting bits with a modulo operation gives a change in the extracted sample whenever the upper bits change. This is far from ideal but probably the best possible result if clock resolution is limited.

Keeping it small

Many RNGs use a cryptographic hash, typically SHA-1, to mix and compress the bits. This is the standard way to distill a lot of somewhat random input into a smaller amount of extremely random output. Seeking a small program, I dispense with the hash. I mix just the input data into a 32-bit word, and output that word when it has enough entropy. The random(4) driver does use a hash, plus other mixing, so a hash is not required here. Details of my mixing are in a later section.

I also do not use S-boxes, although those can be a fine way to mix data in some applications and are a staple in block cipher design. Seeking a small program, I do not want to pay the cost of S-box storage.

While developing this program I looked at an existing “Timer entropy demon”[2], by Folkert van Heusden; it was only at version 0.1 when I checked it. I did borrow a few lines of code from that program, but the approach I took was quite different, so nearly all the code is as well. The timer entropy demon uses floating point math in some of its calculations. It collects data in a substantial buffer, 2500 bytes, goes through a calculation to estimate the entropy, then pushes the whole load of buffered data into random(4).

My program does none of those things; it uses no buffer, no hashing, no S-boxes, and no floating point, only integer operations on a few 32-bit variables. It mixes the input data into one of those variables until it contains enough concentrated entropy, then transfers 32 bits into the random device.

Maxwell's entropy estimation is all done at design time; there is no need to calculate estimates during program operation. A facility is provided for a cautious system administrator, or someone whose system shows poor entropy in testing, to override my estimates at will, using the -p (paranioa) command-line option to make the program use more samples per output. However, even then no entropy estimation is done at run time.

It is possible that my current program's method of doing output – 32 bits at a time with a write() to deliver the data and an ioctl() to update the entropy estimate each time – is inefficient. I have not yet looked at this issue. If it does turn out to be a problem, it would be straightforward to add buffering so that the program can do its output in fewer and larger chunks.

The program is indeed small, just under 600 lines in .c and .h files for maxwell(8) itself, under 750 including test programs. SHA-1 alone is much larger than that, over 7000 lines in the implementation Turbid uses; no doubt this could be reduced, but it could not become tiny. Turbid as a whole is over 20,000 lines and the Cryptlib RNG over 50,000.

The user interface

The program maxwell(8) is a standard sort of Unix program. It comes with a man(1) page as the main documentation and the user can control its operation using command-line options.

It uses the Unix notion that the privileged user root has the special user ID 0. Only root is allowed to change the Linux kernel estimate of available entropy. The program checks whether it is running as root; if so, it updates the kernel entropy estimate after each output unless the -s option (send output to **s**tandard out) has been given.

If the program is run by a non-root user and the -s option is not given, it emits an error message and quits. With -s, it will run as non-root and write test data to standard out. Allowing non-root users to run the program without -s would not be useful since only root can update the kernel entropy counter.

The program provides options which give the user considerable control over its operation. Quoting the manual page:

> Two options take no parameters; these are intended mainly for use in testing:
>
> -s Send output to standard output instead of to /dev/random.
>
> The main intended use is for testing; other options let you control the amount and type of output, and it can be piped into test tools or sent to a file for later analysis.
>
> *A non-root user cannot run the program without this option*. It will just output an error message and quit. With the option, it can be run for testing. It can also be redirected into /dev/random, though no entropy credit will be given.
>
> -<digit>
>
> This runs the given number of loops, where any other combination of options runs at least three. It also omits the extra mixing normally done outside the loop. It is normally used with the -s option.
>
> If maxwell -1 -s | ent shows >= three bits of entropy per byte, then maxwell is safe. That is 12 bits per loop and maxwell uses at least three loops per output.
>
> Several options with numeric arguments allow changing the parameters with which the program runs. If these options are omitted, default values are used.
>
> The numeric arguments are required, and must be in the range 0-99.
>
> -p n Paranoia, affects how much looping is done before each output.
>
> The number of loops done is 2p + 3. Each loop collects sixteen timer samples for entropy.

Default is p = 0 which gives three loops.

-f n Kilobits of data to put out quickly at startup.

At startup, the program produces some output using parameters chosen to quickly fill up the random(4) pool. After that, it changes to more conservative parameters for longer-term use. This parameter controls how much of this first fast data is generated, in kilobits.

Default is 4, enough to completely fill the pool if random(4) is compiled with its default parameters.

-h n   Halt after n kilobits of output.

Default is for the program to continue indefinitely, but limiting the output is useful for testing.

The above options may be combined at will.

Some outputs are intended for use at startup, to quickly fill up the entropy pool. The code generating them uses a smaller delay value (around 50 µs instead of around 100) and always uses only the default three loops rather than whatever the user asks for with -p, so it is considerably faster.

It also claims less entropy per output (16 rather than 30 bits). This does have the effect of making the situation safe even if there are as few as six bits of entropy per 16 samples, since three loops at six bits each gives more than 16 bits output entropy.

However, that is not its main purpose. Even in this fast phase, maxwell(8) is still designed to deliver 32 bits of entropy per output. We tell random(4) there are only 16 so that the pool fills up faster; if the driver estimates that it needs n bits of entropy, then it takes 2n bits of the fast output.

Changes since version 1

maxwell v1 had a different set of options. Here are ways to achieve (approximately) the same effects with the new options:

v1 invocation	**v2 invocation**	**Comment**
maxwell	maxwell -f 0	Default in v2 is to provide fast data; -f 0 disables that
maxwell -f	maxwell -f 4 -h 4	Do 4 K bits of fast data, then quit
maxwell -g	maxwell -f 8 -h 8	Do 8 K bits of fast data, then quit
maxwell -G	maxwell -f 16	Not in my original v1; an improvement from David Jaša Do 16 K bits of fast data, then switch to slower mode
	maxwell maxwell -f 4	I liked David’s idea enough that it is now the default 4 K bits to match the driver’s default 4 K pool size
maxwell -x	maxwell -p 2	
maxwell -y	maxwell -p 4	
maxwell -z	maxwell -p 6	
maxwell -t	maxwell -s -f 0	For testing; produce output indefinitely

Version 2 adds an explicit -s option to send output to standard out; this seems cleaner than making the output file choice depend on -t and -m options.

Some options from version 1 have been dropped entirely:

> -c n Claim, how much entropy do we tell random(4) we are giving it per 32-bit output?
>
> > This is not needed; if input entropy seems low or you just want to be cautious, use -p to increase the rounds instead of reducing the claim. The program now uses fixed claim values, 16 for initial fast data and 30 thereafter.
>
> -d n Delay, microseconds between timer samples.
>
> > Instead of a fixed delay for all output – different for -f, -g, -x, -y, -z and for runs without those options, and optionally settable by the user – the program now uses a random choice from among five delay values for each kilobit of output.
>
> -t Produce an endless stream of test data.
>
> -m Produce a megabit of test output, then quit.
>
> > With -s -f 0 one can get an endless stream of test data, replacing -t. Optionally, use an external tool to get one or more samples of a megabit or any other convenient size, replacing -m.
> >
> > With -s -h <number> one can get up to 99 K bits of test output, and can use -f <number> to control how much of it is from the initial fast phase.

Overall, I hope the new interface is both simpler and more flexible.

Program details

This section describes the program in more detail. It is organised bottom-up, starting with the collection of individual entropy samples, moving up to look at the mixing methods, and finally looking at the global questions of parameter choices.

Sampling the timer

The clock_gettime(2) function fills a structure with two values, one for seconds and one for nanoseconds. I have a generic function for extracting a few bits of entropy from a timer: tmod(m) returns the total timer value (sec*1000000000+nsec) modulo m, taking care to avoid arithmetic overflow.

Using a modulo operation is greatly preferable to using masking to extract some low bits, because it allows all bits in the input to affect the output. Similarly, when only a single output bit is needed, taking the parity is preferable to x&1 because all input bits affect the parity.

There are several functions based on the generic tmod(m) above.

- t5() and t255() just return tmod(5) and tmod(255), the timer value modulo 5 or 255.

- t7() and t31() return tmod(7)+1 and tmod(31)+1, so they always give non-zero values

The ones that return non-zero are more convenient for use in the next level up. Mixing in a non-zero value always produces some change. Of course making them return non-zero neither adds nor loses entropy; it only converts the range from 0 to x-1 to 1 to x.

There are also generic versions of these functions – gmod(m), g5(), g7(), g31() and g255() – using the Unix/Posix standard microsecond timer instead of the nanosecond timer provided by the Linux realtime library. maxwell(8) as released does not use these; they are provided only for convenience if the code needs to be ported.

Basic mixing

The basic operation for mixing a timer sample into a 32-bit word is:

```
      a += ( t31() * mul ) ;
```

An early version used t7(); I changed it to t31() before the maxwell v1 release, and retain this in v2, because this allows for more entropy per sample and there is reason to think there might be enough entropy for this to matter. See discussion under “Testing” below. If there is actually only a bit or two, using t31() does no harm.

I considered using t255() because some tests indicated there might be enough entropy to justify that. However, I am skeptical that timer calls can actually give that much entropy; if they do then t31() throws a little of it away but gives better-mixed output. This seemed a reasonable tradeoff.

The multiplication is there to spread the bits around. In Shannon's terms [14], the t31() call provides confusion and the multiplication provides diffusion. Using a += t31() only a few bits would be changed; with the multiplication, more bits are changed. With t31() always returning non-zero, every sample gives some change.

With the -<digit> testing option, mul = 1. This tests the input entropy, with little mixing. In all other cases, mul is set to the constant MUL. MUL is chosen to spread the input bits out; it is declared as follows:

```
/*
      constant for multiplications

      11, 37, 71 and 41 are 3, 5, 7 and 9 mod 16
      so they all give some mixing in the low bits
      and they each do it differently

      These get multiplied by a value taken mod 31
      so each of the shifted constants can affect
      a different number of bits. e.g. the 1 can
      affect only 5 bits, but 71 can give a value
      up to 31*71 and affect up to 11 bits.
*/
#define MUL (71+(37<<8)+(41<<15)+(11<<19)+(1<<25))
```

Because of the multiplication, a single input sample can affect many bits of the variable a. The shifts are chosen so that the 71 can affect bits 0 to 11, 37 bits 8-18, and so on.

The loop structure

The next level up has a loop that collects 16 samples and mixes them into a 32-bit word:

```
// get 16 samples for entropy
for( k = 0 ; k < 16 ; k++)
{
      usleep(delay) ;
      // mix in a sample
      a += ( t31() * mul ) ;
      // rotate left by two bits
      ...
}
```

The rotation ensures that each sample affects different bits of the variable a.

The get-sixteen-samples loop is inside a get-enough-samples loop:

```
for( j = 0 ; j < loops ; j++)      {
      // get 16 samples for entropy
      for( k = 0 ; k < 16 ; k++ )   {
            ...
            // rotate left by two bits
            ...
            }
      // rotate left one bit
      ...
      }
 }
```

The value of loops is a *critical* security parameter. The default, and the minimum the program allows (except in testing), is three. Analysis below will show that this is a reasonable choice. It is safe if we get 11 or more bits of entropy per 16 samples, since 3*11 > 32. The user can adjust this with the -p (paranoia) command line option. This is discussed in more detail later.

The get-enough-samples loop is inside a do1k() function that outputs one kilobit of data (32 32-bit words). Its main loop is:

```
  for( i = 0 ; i < 32 ; i++) {
      // initialise output variable with a somewhat random constant
      a = sha_c[t5()] ;
      // sample & mix
      for( j = 0 ; j < loops ; j++)      {
            ...
      }
      // output 32 bits
      ...
  }
```

There are two loops using the do1k(unsigned delay, unsigned claim) function. One does the initial fast output, the second all other output. Both use a random delay but they choose from different lists of primes. Each uses a constant entropy claim, 16 for the first loop and 30 for the second.

The setting of the delay for each kilobit block uses a random choice among five primes; for the first loop they have median 53 and mean 51.8, for the second median 101 and mean 99.4. Initialisation of the output variable for each 32-bit output randomly selects one of the five 32-bit constants which SHA-1 uses to initialise its 160-bit state.

This is not essential; I just use random choices instead of setting some fixed value because I can. The extra cost is tiny, it can do no harm, and it might do some good by making an attack more complex. This is just a defense-in-depth trick, not something expected to make the program much more secure.

Output

The section of code that does the actual output (ignoring a few complications) looks like this:

```
// mix thoroughly
a = qht(a) ;
// output 32 bits
if( (ret = write(output, &a, 4)) != 4)
      error_exit("output write failed") ;
// running as root & without -s option
if( demon ) {
     // update entropy estimate
     if( (ioctl(output, RNDADDTOENTCNT, claim)) == -1)
            message("ioctl() failed") ;
}
```

The claim is 16 for the initial fast output and 30 later. Making it less than 32 is a safety measure. The program is designed to give 32 bits of entropy per output, but it tells random(4) it is only delivering 30.

The quasi-Hadamard transform

Just before the output, there is a call to qht(a):

```
// mix thoroughly
a = qht(a) ;
// send 32 bits to /dev/random
```

QHT is a new construction which I call the quasi-Hadamard transform. It is intended to mix its 32-bit input quite thoroughly, to make every bit of the output depend on every bit of input.

The QHT is based on the pseudo-Hadamard transform (PHT) , which was first used in the SAFER ciphers [11] and has since been used in others such as Twofish [12]. A two-way PHT can be described as:

```
x = a+b
y = a + 2b
a = x
b = y
```

but a better implementation is:

```
a += b
b += a
```

which does the operations in place, and eliminates the two intermediate variables and the multiplication by two.

The QHT is just like the PHT except that it uses IDEA [13] multiplication instead of addition; it calculates ab instead of a+b and ab^2 instead of a+2b. Like the PHT, it is invertible; it loses no information.

The preferred implementation, using * for IDEA multiplication, is:

```
a *= b
b *= a
```

This avoids using intermediate variables and having to square b, so it is somewhat more efficient than a naïve implementation.

Because IDEA multiplication makes every bit of its 16-bit output depend on all bits of its two 16-bit inputs, the QHT makes each of its 32 output bits depend on all 32 input bits.

This ensures that the data is well mixed before it is sent to the random(4) driver. The loops acquire enough entropy and do some mixing; this is the final more thorough mixing. It may not actually be necessary since random(4) includes other mixing, and it adds no cryptographic strength since it is an invertible operation, but it seems worthwhile to ensure maxwell output is well-mixed.

Testing

There are some test options built into maxwell(8). Quoting the man(1) page:

```
-<digit>
This runs the given number of loops, where any other combination of options
runs at least three. It also omits the extra mixing normally done outside
the loop.
```

Running maxwell -1 -s tests the bottom-level mixing. It does exactly one of maxwell's loops, collecting 16 samples, using only minimal mixing (a = t31() and the two-bit rotations but no multiplication or qht()), and outputting a 32-bit word to standard out.

There are additional test programs as well.

test.sam tests the entropy sampling. It takes timer samples exactly as maxwell does, but uses none of the mixing code. It just outputs each sample in a byte field. The goal is to test whether there is enough input entropy and this is best done without mixing.

With test.sam -g, the program uses gettimeofday(2) rather than clock_gettime(2), that is it gets data from a microsecond resolution timer rather than nanosecond. maxwell uses the nanosecond timer. This test is there to cover the case of a system lacking the realtime libraries.

The microsecond timer is part of the Posix standard so it should be present on almost any Linux system. If it is not, or if this test gives very poor results, then maxwell will not be usable at all on that system. If it is present and tests OK, using it would require modifying the source and recompiling. The changes would not be extensive; just replace various t*() function calls with g*() equivalents and possibly adjust the line #define MIN_LOOPS 3 to compensate for lower input entropy.

The test programs. test.mix.c and test.qht.c test parts of maxwell(8) which test.sam and maxwell -s ignore. They collect no entropy samples at all; they just use a counter in place of sampling.

test.mix.c just makes three loops of 16 samples each where a sample is a counter modulo 31. ent(1) reports 6.4 bits per output byte for this, remarkably high for something that repeats itself every 48*31 output words. This indicates that the maxwell(8) mixing using multiplications and shifts is effective.

test.qht.c tests the final mixing with qht(). The main loop is:

```
// test outer loop of maxwell
for( i = 0 ; i < LIMIT ; i++ ) {
      // all the sampling replaced
      a = qht(i) ;
      // write to stdout
      write(1, &a, 4) ;
}
```

In principle, this program has no entropy at all, just the counter which is completely predictable. However, ent(1) reports 7.95 bits per output byte. This is an excellent result; it indicates that maxwell's mixing is thorough enough to give apparently excellent randomness even when the input varies little. A counter plus the qht() operation provide random output approximately as a counter plus a block cipher would.

Running the tests

To test, be sure the entropy measurement program ent(1) is installed, then type make test, which compiles maxwell then runs:

```
./maxwell -s -h 64 -f  0 | ent | grep Entropy
./maxwell -s -h 64 -f 64 | ent | grep Entropy
```

This produces 64 K bits of output, first from the main loop and then from the fast startup loop. Results above 7.5 bits/byte (30 per output) in the first test and 5.33 bits/byte (16 per output) in the second indicate that maxwell(8) should be safe, though this not an infallible test.

I would recommend that anyone building maxwell(8) for their own system or for distribution run at least these simple tests as a sanity check. On my system, both give about 7.975 bits/byte which gives me some confidence that the program is working correctly.

Standalone test programs are also provided to test individual parts of the program – sampling, mixing and so on. For these, do make full_test which compiles the test programs, creates a results directory if you do not already have one, then runs:

```
maxwell -1 -h 64  | ent > results/max.1.out
test.sam          | ent > results/raw.out
test.sam -g       | ent > results/usec.out
test.qht          | ent > results/qht.out
test.mix          | ent > results/mix.out
test.parity       | ent > results/parity.out
test.parity -g    | ent > results/par.g.out
grep Entropy results/*.out > test.summary
```

There are additional, much more expensive, tests which are invoked separately with make diehard. These are described separately later.

We need eleven or more bits of entropy per loop for maxwell(8) to be secure with three loops. This means it is safe if:

- max.1.out shows three bits per output byte, which is twelve per loop
- raw.out shows at least .7 bits per sample, which gives .7*16 ~= 11 bits per loop
- parity.out shows at least 5.5 bits per byte, equivalent to 11 per loop

A modified version using the Unix standard microsecond timer, rather than the nanosecond timer provided with the realtime libraries, would be safe if:

- usec.out shows at least .7 bits per sample, which gives .7*16 ~= 11 bits per loop

Testing on my current system, a laptop with an Intel CPU and Lubuntu; my results are:

Test program	**Output file**	**Bits needed per test byte**			**Test result**
		Input data	**Output data**	**Mixing**	
maxwell -1	max.1.out		8/3 = 2.67		7.99
test.sam	raw.out	32/48 = 2/3			4.95
test.sam -g	usec.out	2/3			4.95
test.parity	par.t.out	8*2/3 = 5.33			7.82
test.parity -g	par.g.out	5.33			4.36
test.mix	mix.out			anything	6.39
test.qht	qht.out			anything	7.95

The last two results show that maxwell's mixing works rather well. Those programs use no timer samples so they have no genuine entropy at all; they just take a counter modulo 31 and run that through parts of the maxwell mixing. Despite that, they are able to fool ent(1); the output looks random.

Overall, these results appear to show that maxwell(8) has more entropy available than it needs, except in one case – using test.parity -g which weakens the algorithm in two ways, using the microsecond timer and taking its parity (only one bit of entropy per sample is possible) rather than using a modulo 31 calculation (perhaps near five bits).

However, it is necessary to remember that ent(1) can test only the *apparent* entropy of the sequence it is given. The RFC has:

> Statistically tested randomness in the traditional sense is NOT the same as the

> unpredictability required for security use. [1]

A test program such as ent(1) should give large entropy numbers for the output of a good pseudorandom generator; that is the sort of thing it was designed to test. However, the actual entropy of such a generator cannot exceed that of its key. We therefore know that, for a well-mixed sequence, ent(1) can sometimes give readings higher than the true entropy. That may be the case in any of these tests, so results must be interpreted with some caution.

Even with that caveat, these results are reassuring. When ent(1) says we are getting almost five bits of input entropy per sample, it seems plausible to assume we are getting at least one. With 48 samples for 32 output bits, we only need two thirds of a bit per sample to be safe.

My results do not contradict those in the inherent randomness paper [10], only extend them somewhat. They used masking to extract the low bit of each sample, making it impossible for their tests to find more than one bit per sample. My tests using modulo operations expose more entropy, if it is there.

Non-Intel CPUs

If your system is anything but a bog standard PC, it is very strongly recommended that you run these tests to check that you are indeed getting enough entropy, at least 11 bits per 16 samples. Even on a PC, you might as well run the tests.

I have not tested on CPUs other than those I have had to hand which have all been Intel. I appeal to anyone who can test on a very different CPU to do so and send me the results. I would particularly like to hear from anyone who gets results, on any CPU, significantly different from mine. The email address is at the top of the paper.

If your system tests poorly

If the tests give problematic results on your system, using the -p option provides a temporary fix. A longer-term solution is to increase the constant MIN_LOOPS in the source code, recompile, and install the new version. This will ensure that, even when invoked without -p, the program does enough loops to be secure on your system.

Diehard tests

The most thorough test is obtained with make diehard. This runs the command:

```
maxwell -s | dieharder -a -g 200 -k 2 -Y 1 > results/dh.max.out
```

The program dieharder(1) is a thorough entropy tester; the -a option says to apply all tests and -g 200 means to expect a stream of binary input. The other options make it a very stiff test.

The -s option to maxwell(8) makes it run indefinitely and send output to standard out; this is necessary because dieharder(1) needs a great deal of input. This is an extremely slow and expensive test. On my machine, a lightly loaded 3 GHz desktop, it takes days.

Analysis

This section discusses the program design in more detail, dealing in particular with the choice of appropriate parameter values.

How much entropy?

The inherent randomness paper [10] indicates that almost a full bit of entropy can be expected per timer sample. Taking one bit per sample and packing eight of them into a byte, they get 7.6 bits per output byte. Based on that, we would expect a loop that takes 16 samples to give just over 15 bits of entropy. In fact we might get more because maxwell(8) uses a modulo operation instead of just masking out the low bit, so getting more than one bit per sample is possible.

I designed the program on the assumption that, on typical systems, we would get at least 12 bits per 16 samples, the number from the inherent randomness [10] paper minus something for safety. This meant it needed three loops to be sure of filling a 32-bit word, so three is the default.

I also provide a way for the user to override the default where necessary with the -p (paranoia) command-line option. However, there is no way to get fewer than three loops, so the program is always safe if 16 samples give at least 11 bits of entropy. The tradeoffs are as follows:

-p	loops	Entropy needed per 16 samples		
(paranoia)	2p + 3	For 32 bits out	For 30-bit claim	For 16-bit claim
0	3	11	10	6
1	5	7	6	Not applicable Fast output phase always uses only three loops
2	7	5	5	
3	9	4	4	
4	11	3	3	
5	13			
6	15	3	2	
7	17	2	2	
...				
15	33	1	1	

All these entropy requirements are well below the 15 bits per 16 samples we might expect based on the inherent randomness paper [10]. They are also far below the amounts shown by my test programs, described in the previous section. I therefore believe maxwell(8) is, at least on systems similar to mine, entirely safe with the default three loops; using -p should be unnecessary on most systems.

I find it impossible to imagine circumstances where more than -p 6 would be needed, but the program accepts any number up to 99. Users may have different systems than any I am familiar with, or perhaps just a better imagination.

Attacks

The Yarrow [7] paper gives a catalog of possible weaknesses in a random number generator. I shall go through each of them here, discussing how maxwell(8) avoids them. It is worth noting, however, that maxwell(8) does not stand alone; its output is fed to random(4), so some possible weaknesses in maxwell(8) might have no effect on overall security.

The first problem mentioned in [7] is “Entropy Overestimation and Guessable Starting Points”. They say this is both “the commonest failing in PRNGs in real-world applications” and “probably the hardest problem to solve in PRNG design”.

My detailed discussion of entropy estimation is above. In summary, the outputs of maxwell(8) have 32 bits of entropy each if each timer sample gives two thirds of a bit. The Inherent Randomness paper [10] indicates that about one bit per sample can be expected and my tests indicate that more than that is actually obtained. Despite that, we tell random(4) that we are giving it only 30 bits of entropy per output, just to be safe.

There are also command-line options which allow a system administrator to overrule my estimates. If maxwell is thought dubious with the default parameters, try maxwell -p 6 or some such. That uses 16*(3 + 2*6) = 240 timer samples and claims 30 bits of output entropy, so it is secure if samples average an eight of a bit each.

There is a “guessable starting point” for each round of output construction; one of five constants from SHA-1 is used to initialise the sample-collecting variable. However, since this is immediately followed by operations that mix many samples into that variable, it does not appear dangerous.

The next problem [7] mentions is “Mishandling of Keys and Seed Files”. We have no seed file and do not use a key as many PRNGs do, creating multiple outputs from a single key. Our only key-like item is the entropy-accumulating variable, that is carefully handled, and it is not used to generate outputs larger than input entropy.

The next is “Implementation Errors”. It is impossible to entirely prevent those, but my code is short and simple enough to make auditing it a reasonable proposition. Also, there are test programs for all parts of the program.

The next possible problem mentioned is “Cryptanalytic Attacks on PRNG Generation Mechanisms”. We do not use such mechanisms, so they are not subject to attack.

Of course, our mixing mechanism could be attacked, but it seems robust. The QHT is reversible, so if its output is known the enemy can also get its input. However, that does not help him get the next output. None of the other mixing operations are reversible. Because the QHT makes every bit of output depend on every bit of its input, it appears difficult for an enemy to predict outputs as long as there is some input entropy.

The next attacks discussed are “Side Channel Attacks”. These involve measuring things outside the program itself – timing, power consumption, electromagnetic radiation, … – and using those as a window into the internal state.

It would be quite difficult for an attacker to measure maxwell's power consumption independently of the general power usage of the computer it runs on, though perhaps not impossible since maxwell's activity comes in bursts every 100 µs or so.

Timing would also be hard to measure, since maxwell(8) accepts no external inputs, its only output is to the kernel, and the delay is randomly chosen anew for each K bit block of output. Data-dependent multiplications have been shown to lead to timing attacks on some ciphers since on some architectures the timing of the operation is affected by the data used so, given enough timing data, an enemy may infer the key. maxwell(8) does use data-dependent multiplications in tmod(), in a = tmod * MUL and in qht(), but this does not seem dangerous since there is no key to be inferred and an attacker has no way to measure the timing unless he or she already has root privileges.

A Tempest type of attack, measuring the electromagnetic radiation from the computer, may be a threat. In most cases, a Tempest attacker would have better things to go after than maxwell(8) – perhaps keyboard input, or text on screen or in memory. If he wants to attack the crypto, then there are much better targets than the RNG – plaintext or keys, and especially the private keys in a public key system. If he does go after the RNG, then the state of random(4) is more valuable than that of maxwell(8).

However, it is conceivable that, on some systems, data for other attacks would not be available but clock interactions would be visible to an attacker because of the hardware involved. In that case, an attack on maxwell(8) might be the best available attack. If Tempest techniques could distinguish clock reads with something close to nanosecond accuracy, that would compromise maxwell(8). This might in principle compromise random(4) if other entropy sources were inadequate, though the attacker would have considerable work to do to break that driver, even with some known inputs.

The next are “Chosen-Input Attacks on the PRNG”. Since maxwell(8) uses no inputs other than the timer and uses the montonic timer provided by the Linux realtime libraries, which not even the system administrator can reset, direct attacks on the inputs are not possible.

It is possible for an attacker to indirectly affect timer behaviour, for example by accessing the timer or running programs that increase system load. There is, however, no mechanism that appears to give an attacker the sort of precise control that would be required to compromise maxwell(8) – this would require reducing the entropy per sample well below one bit.

The Yarrow paper [7] then goes on to discuss attacks which become possible “once the key is compromised”. Since maxwell(8) does not use a key in that sense, it is immune to all of these.

Some generators allow “Permanent Compromise Attacks”. These generators are all-or-nothing; if the key is compromised, all is lost. Others allow “Iterative Guessing Attacks” where, knowing the state at one time, the attacker is able to find future states with a low-cost search over a limited range of inputs, or “Backtracking Attacks” where he can find previous states. However, maxwell(8) starts the generation process afresh for each output; the worst any state compromise could do is give away one 32-bit output.

Finally, [7] mentions “Compromise of High-Value key Generated from Compromised Key”. However, even if maxwell(8) were seriously compromised, an attacker would still have considerable work to do to compromise random(4) and then a key generated from it. It is not clear that this would even be possible if the system has other entropy sources, and it would certainly not be easy in any case.

Resources and speed

As intended, this program needs remarkably few resources. On my Intel/Ubuntu machine the executable is just over twelve K bytes.

The program does not use much CPU. It spends most of its time sleeping; there is a usleep(delay) call before each timer sample, with delays around 50 µs in the initial fast phase and 100 µs later. When it does wake up to process a sample, it does only a few simple operations.

The rate of entropy output is adequate for many applications; I have argued above that a few hundred bits per second is enough on most systems. This program is capable of about an order of magnitude more than that.

In the main output phase without -p, there are 48 usleep(delay) calls between outputs, at an average 99.4 µs each so total delay averages 4.77 ms. Rounding off to 5 ms to allow some time for calculations, we get an estimate that the program can do 200 32-bit quantities per second, just over six K bit/s. In the initial fill-up-the-pool phase, the delay is shorter and estimated output rate over 10 K bit/s. With -p the rate slows down in the main phase, but even with -p 6 we might still get over one K bit/s.

Actual test times are somewhat lower. On my system (a laptop with a 2.16 GHz Pentium R CPU) doing *time maxwell -s -h 64 -f 0 > /dev/null* gives a real time of 16.24 seconds for 64 K bits of output, or just under 4 K bit/s. With -p 6 the time is 1m20s, and rate about 800 bit/s.

On a busy system, the program may be delayed because it is timeshared out, or because the CPU is busy dealing with interrupts. We need not worry about this; the program is fast enough that moderate delays are not a problem. If the system is busy enough to slow this program down significantly for long enough to matter, then there is probably plenty of entropy from disk or net interrupts. If not, the administrator has more urgent things to worry about than this program.

The peak output rate will rarely be achieved, or at least will not be maintained for long. Whenever the random(4) driver has enough entropy, it causes any write to block; the writing program is forced to wait. This means maxwell(8) behaves much like a good waiter, unobtrusive but efficient. It cranks out data as fast as it can when random(4) needs it, but automatically waits politely when it is not needed.

Conclusion

This program achieves its main design goal: it uses minimal resources and provides high-grade entropy in sufficient quantity for many applications. Also, the program is simple enough for easy auditing.

The version 1 user interface was a first cut, usable but not ideal. The v2 interface is better, simple but providing reasonable flexibility.

References

[1] Eastlake, Schiller & Crocker, Randomness Requirements for Security, June 2005, http://tools.ietf.org/html/rfc4086
[2] Timer entropy demon web page http://www.vanheusden.com/te/
[3] Wikipedia page on Johnson-Nyquist noise http://en.wikipedia.org/wiki/Johnson-Nyquist_noise
[4] Turbid web page http://www.av8n.com/turbid/
[5] A. Seznec, N. Sendrier, "HAVEGE: a user level software unpredictable random number generator" http://www.irisa.fr/caps/projects/hipsor/old/files/HAVEGE.pdf
[6] HAVEGE web page http://www.irisa.fr/caps/projects/hipsor/
[7] J. Kelsey, B. Schneier, and N. Ferguson, Yarrow-160:
Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator, SAC 99 http://www.schneier.com/yarrow.html
[8] Cryptlib web page http://www.cs.auckland.ac.nz/~pgut001/cryptlib/
[9] Peter Gutmann, Software Generation of Practically Strong Random Numbers, USENIX Security Symposium, 1998
http://www.usenix.org/publications/library/proceedings/sec98/gutmann.html
[10] McGuire, Okech & Schiesser, Analysis of inherent randomness of the Linux kernel, http://lwn.net/images/conf/rtlws11/random-hardware.pdf
[11] James L. Massey, SAFER K-64: A Byte-Oriented Block-Ciphering Algorithm, FSE '93
[12] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson
Twofish: A 128-Bit Block Cipher, First AES Conference
http://www.schneier.com/papertwofishpaper.html
[13] Xuejia Lai (1992), "On the Design and Security of Block Ciphers", ETH Series in Information Processing, v. 1
[14] Claude Shannon, “Communication Theory of Secrecy Systems”, Bell Systems Technical Journal, 1949, http://netlab.cs.ucla.edu/wiki/files/shannon1949.pdf
[15] Web page for haveged(8) http://www.issihosts.com/haveged/
[16] Gutterman, Pinkas & Reinman “Analysis of the Linux Random Number Generator”, http://eprint.iacr.org/2006/086