The *maxwell(8)* random number generator

Sandy Harris
sandyinchina@gmail.com
sandy.harris@sjtu.edu.cn

Abstract

I propose a daemon process for use on Linux. It gathers entropy from timer calls, distills 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 varies with the parameters chosen; with the defaults it is about six kilobits per second, which is enough for many applications.

1 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.

1.1 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 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, SSH/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; however, 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.

1.2 Problem statement

The kernel generator provides an interface that allows an external program to provide it with additional entropy, to prevent any potential entropy 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 daemon program which I believe solves this problem. I wanted a name distinct from the existing "Timer entropy daemon" [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.

2 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 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.

2.1 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 be used for crypto, insisting on a hardware RNG as part of your specification is completely reasonable.

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 process, but this is not a very 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. Adding a second good source of entropy—*maxwell(8)*, Turbid or HAVEGE—makes an attack via RNG subversion utterly implausible.

2.2 Turbid

John Denker's Turbid—a daemon 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 often 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 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 most 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.

2.3 HAVEGE

The HAVEGE (HArdware Volatile Entropy Gathering and Expansion) RNG gathers entropy from the internal state of a modern superscalar processor. There is a daemon 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 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 very 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 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's state includes internal processor state not knowable by an opponent and moreover it is continuously updated, so it appears to meet this criterion.

The *haveged(8)* daemon 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 carefully 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.

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.

2.4 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 of course 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.

3 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 is high.

3.1 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 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 fully trusted. Taking a broader view, any well-designed generator 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 HAVEGE runs on your CPU, or you have a sound device free for Turbid—that is the clearly the generator to use. Any of these can give large amounts of high-grade entropy for little resource cost. If two of them are available, consider using both.

If none of those is easily available, the choice is more difficult. It is possible to use *maxwell(8)* in all cases, but using the Cryptlib RNG or adding a device for Turbid should also be considered. In some situations, using an external hardware RNG is worth considering as well.

3.2 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
- 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 kilobits a second is clearly enough

There are three main applications:

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.

With the *-f* or *-g* option, *maxwell(8)* runs faster and stops after a fixed amount of output. This is suitable for filling up the entropy pool at boot time, or before some randomness-intensive action such as generating a large PGP key.

maxwell(8) can be used even on a very limited systemi—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.

More than one copy of *maxwell(8)* can be used. The computer I am writing this on uses *haveged(8)* with *maxwell -z* (slow but sure) as a second entropy source and *maxwell -g* for initialisation. This is overkill on a desktop system—probably any of the three would be enough. However, something like that might be exactly what is needed on a busy server.

4 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 that not only is this is achievable but 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.

4.1 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. However, 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 a system supports 300 connections and re-keys each of them every 20 minutes, then it will do 900 re-keys an hour, one every four seconds on average. In general, session keys need only a few hundred bits and can get those from */dev/urandom*. Even if each re-key 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.

This would indicate that *maxwell(8)* needs to produce a few hundred bits per second. In fact, it gives an order of magnitude more, a few K bits per second. Details are in the "Resources and speed" section.

4.2 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)*, giving 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, command-line options allow the administrator to increase the number of samples per output (*-p*) or to reduce the amount of entropy claimed (*-c*) 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 problematic. However, extracting bits with a modulo operation gives a change in the extracted sample whenever the upper bits change.

4.3 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. Details of the 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.

In developing this program I looked an existing "Timer entropy daemon" [2] developed by Folkert van Heusden. It is only at version 0.1. 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 daemon 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.

maxwell(8) uses no buffer, no hashing, and no S-boxes, only a dozen or so 32-bit variables in various functions. It mixes the input data into one of those variables until it contains enough concentrated entropy, then transfers 32 bits into the random device. The 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 command-line options, *-p* (paranioa) to make the program use more samples per output or *-c* (claim) to change the amount of entropy it tells *random(4)* that it is delivering. However, even then no entropy estimation is done during actual entropy collection; the user's changes are put into effect when the program is invoked.

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, under 500 lines in the main program and under 2000 overall. SHA-1 alone is 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.

5 Program details

The source code for this program is available from `ftp://ftp.cs.sjtu.edu.cn:990/sandy/maxwell/`. The archive includes a more detailed version of this paper, covering the command-line interface, the internal design of the program, and testing methodologies for evaluating the quality of the output.

6 Analysis

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

6.1 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

-p (paranoia)	Options (other than -p)	Loops 2p + 3	Entropy needed per 16 samples	
			for 32 bit out	for 30 bits claimed
0	*No options*	3	11	10
1		5	7	6
2	-x	7	5	5
3	-y	9	4	4
4	-z	11	3	3
⋮		⋮	⋮	⋮
7		17	2	2
⋮		⋮	⋮	⋮
15		33	1	1

Table 1: The *-p* (paranoia) option

-p (paranoia)	Options (other than -p)	Loops 2p + 3	Entropy needed per 16 samples	
			for 32 bit out	for 30 bits claimed
0	-f, -g	3	11	6

Table 2: The *-f* or *-g* options

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 trade-offs are shown in Table 1. With the *-f* or *-g* options, the claim is reduced, as shown in Table 2.

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.

In my opinion, setting *-p* higher than four is unnecessary, even for those who want to be cautious. However, the program accepts any number up to 999.

6.2 Attacks

The Yarrow paper [7] 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 here. 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(8)* is thought dubious with the default parameters, try *maxwell -p 3 -c 20* or some such. That is secure if 144 timer samples give 20 bits of entropy.

There is a "guessable starting point" for each round of output construction; one of five constants borrowed from SHA 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 mentioned in [7] 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. *random(4)* does use such mechanisms, but they are designed to resist cryptanalysis.

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 and its only output is to the kernel.

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 possibility for the attacker. If an attacker using Tempest techniques could distinguish clock reads with 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 monotonic timer provided by the Linux real-time 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.

The program does not use much CPU. It spends most

Option	Delay	Samples	msec per output	K bit/sec
Default	97	48	5	∼6
-f, -g	41 or 43	48	2	∼16
-x	101	112	11.3	∼2.8
-y	103	144	14.8	∼2.1
-z	107	176	18.8	∼1.6

Table 3: Output rates

of its time sleeping; there is a *usleep(delay)* call before each timer sample, with delays generally around 100 μs. 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. With the default parameters there are 48 *usleep(delay)* calls between outputs, at 97 μs each so total delay is 4.56 ms. Rounding off to 5 ms to allow some time for calculations, we find that the program can output up to 200 32-bit quantities—over six kilobits—per second. Similar calculations for other parameter combinations are shown in Table 3.

Of course, all of these are only approximate estimates. Testing with *dieharder(1)* and a reduced delay shows 367 32-bit rands/sec or 11.7 Kbits/sec, showing that these figures are not wildly out of whack but are likely somewhat optimistic.

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.

7 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 user interface is at least a decent first cut, 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 daemon 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. /em 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/paper-twofishpaper.html

[13] Xuejia Lai. On the Design and Security of Block Ciphers. *ETH Series in Information Processing, v. 1, 1992*.

[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