Principles and Implementation of ESRNGs - Embarrassingly Simple Random Number Generators for GNU/Linux

Nicholas Mc Guire
OpenTech EDV Research GmbH
Bullendorf, Austria
der.herr@hofr.at

Abstract

Generating random numbers by traditional means, that is harvesting asynchronous events from peripheral devices via interrupts, is limited on may embedded systems and for some even completely impossible. This not only raises security issues, but it increasingly is also a problem for algorithms that depend on unbiased random numbers (e.g. monte carlo methods). For these deeply embedded systems the solution has typically been to used "good" pseudo random number generators and hope (in the security case) that the adversary will not be able to figure out the seed and the PRNG algorithm and thus not be able to predict sequences. The alternative was a relatively expensive TRNG hardware extension.

In the past decades computer science has extensively studied the jitter and latency properties of computer systems. A generally observable trend is for the jitter/latency distribution to manifest it self as distribution rather than as discrete peaks in the spectrum. These distributions are only in part due to peripherals active or the asynchronous nature of co-processors (e.g. DMA,GPUs, etc.) a significant part of the distribution must be attributed to the inherent non-determinism of modern super-scalar CPUs them selves. Pairing this inherent non-determinism, that does not depend on peripheral activities, with methods for harvesting entropy seems a natural option for deeply embedded systems to provide reliable random number generation - including use in cryptographic applications.

The concept of generating random numbers by using "non-deterministic" software constructs is it self not new - attempts to utilize the undeterminedness of wakeup order from a semaphore have been published in the past - the overall literature on the issue is though scant. In the past years we have developed three different entropy extractors. All of them have one thing in common - the method used is actually trivial and thus we dub this class of random number generators Embarrassingly Simple Random Number Generators (ESRNGs). The three concepts that have been implemented to date are:

1st ESRNG (x86 only): was based on the reading of the TSC being physically non-deterministic, even with disable interrupts and cache-hot code, the variance was very small though an extraction was never the less possible. This ESRNG though mandated operation with disabled interrupts to ensure that it was not possible to induce patterns on the bit extraction.

2nd ESRNG: The second ESRNG approach was to use a general race condition, a simple unprotected global variable and count the occurrence of a race as a 1 and a non-occurrence as a 0. The outcome would depend on the length of the trial loop and this trial loop was then used to dynamically adjust the ESRNG to varying load situations. While this works we have, to date, not been able to auto-calibrate the parameters needed for the control loop.

3ed ESRNG: One of the drawbacks of the 2nd ESRNG was the complexity of the control loop and the fact that the entropy harvested was very limited as it was based on a single bit per race trial (some experimental extensions did show that more could be extracted but that does not remedy the principle limitations of the approach) so for the 3ed ESRNG we tried to model not a single bit "ball on the nail" type RNG but extended this to logically form a Galton board - that is a state preserving (or partially state preserving) approach.

In this presentation we will not only outline the principles behind the entropy extraction mechanism and present currently available (while still limited) data but also introduce the actual code that is currently undergoing testing for the 3ed ESRNG, on a set of platforms ranging from uniprocessors running a 2.4.X kernel to 8 core systems running the latest 3.X kernels and 3.4.X preempt-rt kernels. this still limited

spectrum of hardware and kernels never the less gives us the confidence that the approach is suitable and potentially can be generalized. The prime goal of the presentation is to present the methodology and the design of these random number generators and discuss in what form this could be integrated at kernel level to provide reliable random number generation for deeply embedded systems.

1 Introduction

Random number generation has been an issues ever since science discovered that some physical processes are best approximated by stochastic models. While the quality of random numbers for modeling was not that critical, and in many cases pseudo random numbers suffice, security put stringent demands on random number generators in the sense that any, even partial predictability could be used by an adversary in decrypting information. With this demand on quality of randomness comes the demand for random number generation in computers. Many methods have been devised to extract entropy from physical interaction of computer systems with their environment - notably based on interrupts. Alternative dedicated devices, known as true random number generators (TRNG) have been developed that extract entropy from a truly stochastic physical process, be it nuclei decay, thermal noise or quantum effects.

With all these sources of randomness - why bother with a further solution ?

One domain that has a real problem extracting sufficient entropy for the generation of random numbers are deeply embedded systems with either no or only highly deterministic peripherals (CAN or 1553 bus) and thus no source of entropy on the interrupt side - at the same time this class of devices typically is not able to use high-cost dedicated TRNGs.

The requirement for a TRNG though is simply a source of physical entropy - and in this paper we develop a simple analogy to physical models used for random number generation and show that these can be implemented in software on any contemporary CPU. This allows to extract the inherent entropy of modern super scalar CPUs with the help of quite trivial code constructs based on a well designed race-condition (this one in fact is truly a feature not a bug).

Providing an effective entropy extraction utility for a general purpose OS like GNU/Linux is though potentially not only of interest for deeply embedded systems but also for the general OS user as this extraction method potentially allow using a local source that is hence un-observable from the outside, potentially allowing to improve security of common desk-top systems.

As the random number generator is based on trivial code - one could claim it is based on a common bug - we have named it embarrassingly simple random number generator (ESRNG) and as this is the third such generator we declare this principle to constitute a separate class of random number generators. This ESRNG is implemented in user space and runs as an unprivileged process. While our first implementation required root privileges and was a X86 specific hack, this solution is hardware agnostic and in principle also in no way GNU/Linux specific.

This is work-in-progress and has not yet been certified to be cryptographically secure by any suitable authority, we will focus on the principle here, and introduce a practical implementation as proof-of-concept. More work needs to be done - possibly very different code needs to be developed - but the principle we believe is sound never the less.

2 Background

Generation of random numbers has been a long standing problem, starting from impressive empirical runs of 100.000 throws of dices ([8]) to physical random number generation that were then recorded in extensive tables of 1 million random numbers ([7] all the way to random numbers based on quantum physics effects [6] in relatively recent works on Carbon Nanotube Device (CNT) for random number generation. There are of course a number of TRNGs based on slightly more conservative technology available as well - never the less these special purpose devices are not only expensive but simply limited due to availability - notably in embedded systems - after all who would like a cobalt 58 source in his mobile phone just to be really secure....

The alternative, at least for GPOS that have sufficient asynchronous interrupt sources, was to use external (external to the computer system) events and the non-determinism of when they would occur to feed and entropy pool of the OS. The underlying assumption being that these external events are a. independent and b. randomly distributed.

In GNU/Linux this is provided to the user through */dev/random* - while of high quality it is of quite limited capacity - a few bytes per second at best a few 10 bytes per second. Aside from this performance issue what is with systems that are deeply embedded and don't have any reasonably random asynchronous events but still need random numbers for security reasons or for algorithmic purposes ? Where could we get the entropy from ?

Any kernel programmer (or programmer of concurrent code) knows how hard it can be to reproduce a race condition - some of them only manifesting them selves within runs of weeks or month - with other words reproducing a very low frequency "random" process - even though the cause is a perfectly deterministic design bug. The question is if this is true randomness or only perceived randomness - like PRNGs that humans would generally also consider to be truly random if naively inspected - that is without mathematical rigor.

With security demands not only on servers and desktop but also on mobile devices and deeply embedded (headless) devices the question of random numbers for cryptographic purposes has been raised many times in the past decades. There have been many answers from pseudo random number generators (PRNG [5]) to elaborate variations of special purpose devices ranging from FPGA based TRNGs to quantum theory TRNGs utilizing carbon nanotube technologies.

While all of these special devices may have their merits they have a common short coming - availability on embedded devices and cost. On the other hand PRNGs have there limits with respect to cryptographic strength. Alternatives have been noted in literature [4] based on the perceived random nature of race conditions in modern operating systems - though the author concluded that this would only be a nice PRNG but not a true random number generator. While the semaphore wake up order does depend on some OS settings as well as system load conditions this does not make it a PRNG per-se - the question is what is the cause for the undetermined wakeup order ? In [3] we argued that there actually is an inherent, true randomness, in complex hardware software systems and in this paper we will present first implementations of entropy extraction utilizing the inherent randomness of modern CPUs.

Based on analyzing data generated on NN COTS systems running GNU/Linux we believe to have good evidence that these events may well be based on true randomness and even more, that any moder superscalar CPU is actually a TRNG with additional computational capabilities - we simply did not yet find the right way to extract the enormous entropy source of these CPUs.

In this paper we introduce the concept behind the ESRNGs (Embarrassingly Simple Random Number Generators) and argue that they not only are in fact true random number generators but also that they can provide the necessary source of random numbers in modern systems. The code is trivial at its core - the problems for building an extractor not yet resolved, at best we can claim to be able to show that it is possible - more work is to be done.

We briefly establish a equivalent physical model of what this strange code is doing and then step-by-step transform this model into code on a computer. Finally we show some of the problems that this model has with systems that don't operate in a steady state and how to mitigate them at an empirical level, which is shown by some of the test results. We end with some concluding remarks on random number generation and hope that this will be an inspiration to investigate in more detail what capacity modern CPUs have for the generation of random numbers.

3 Controlling entropy extraction

basic concept is simple. Lets start with a example. Take a system that only consists of a timer and a facility to read the timer. We now initialize two tasks

- Task A: reads the timer in X seconds from NOW
- Task B: loops Y times on some code and then terminates

if A completes before B we call it a 0 else a 1

Now lets take a perfect system where every instruction takes 1 cycle and the timer code is jitter free. In such a system we can now determine Y so that they take exactly the same time to execute. In fact task A is also simply a loop, just that it is in hardware. So we have a system that fires the timer exactly at the same time that the task B terminates its loop. So where is the randomness ?

We let both tasks emit a message "A is done" respectively "B is done" and we now use a single shared resource to emit this message - the console. Now no mater how precise the information about task A and B are, the console output will either start with A or

B, it can't start with both ! So despite the perfection of the underlying system, being jitter free and constant cycle time, it will emit only one message first - which one ? The maybe seemingly paradox answer is - if the system is perfectly jitter free it will randomly emit A or B first there is no way to control it. This is effectively the perfect ball on the nail equivalent in computer science - just with one subtle difference. In mechanical systems perfect balls dropped on a perfect nail could infinitely stack - never dropping a single ball left or right (well in Newtonian mechanics that doable - not in real life).

In computers this option does not exist if we serialize on a shred resource like the console in the above case - every ball will go left *or* right - no stacking on computer consoles.

What does all this have to do with a control loop - we just need a perfect computer and our entropy extractor is done ? Well that's the problem, computers are not perfectly deterministic, so we need to force this determinism or at least approximate it by a either a feedback controller (which turned out to be very hard to design) or by statistically enforcing the occurrence of the race condition and selecting outputs based on the occurrence of an actual race event - kind of the equivalent of a feed-forward controller. The first option, the feedback controller, was what we tried in ESRNG version 2 but still do not have satisfactory solutions to it - we will denounce this option (at least until we understand enough to implement the control loop calibration..). The second option turns out to be very simple to implement though with a significantly lower extraction rate, this second version esrng2 is what will be covered here in more detail.

Note on naming - the first ESRNG was called trng.c so that is why the third version of the ESRNG is called esrng2.c.

3.1 Generators ?

Random numbers are never generated (at least not by software on this planet) - Van Neumann stated this in a very clear form in the early 50s by saying:

"Anyone who considers arithmetic methods of producing random digits is, of course, in a state of sin" [Reportedly said by Von Neumann at a conference on Monte Carlo methods in 1951]

In this sense there are no random number generators - all we have is entropy extraction tools (hardware and software) that allow us to associate numerical values with the state of a stochastic process and so represent the entropy of this stochastic process as a series of random values - effectively any system has some form of inherent randomness so every system is a random number generator - all we need is the suitable extraction utility. In this paper we believe that we have shown that such a utility can be constructed for a general purpose OS like GNU/Linux.

4 Practical implementation

The goal of this paper is to show how to practically write random number generators which is actually quite simple - so in this section we start with a simple race and extend it to a full features ESRNG by stepwise refinement.

4.1 background

To design a random number generator in software is actually quite simple, what you need is a operation sequence that is perfectly predictable *if* run single threaded and then simply run it concurrent sharing some object. Lets start with about the simplest possible RNG in software possible.

```
count++;
```

if we look at what this really is doing it is in fact

```
read       RAM_location,Register
increment  Register
write      Register,RAM_location
```

As soon as we run this concurrent we have at least 2 opportunities for race occurrence.

```
Task A           Task B
read  mem,reg
                 read  mem,reg
inc   reg        inc   reg
write reg,mem
                 write reg,mem
```

The read has to be in some order as Task A and B can't actually read from the memory location of mem physically at the same time, caches and the like do not change this in any way - it may change probabilities of observing races though. And at some point the tasks will write back results and again some task has to be first to physically write back the value to the memory location. Now if the initial value of the memory location was 0 and we get the above execution order the final result will be that both tasks incremented the register from 0 to 1 and wrote

back 1, with task B overwriting task As 1 with 1 - not very exciting race condition. The "excitement" comes from the inability to actually determine the order of execution - so an alternative execution path would be:

```
Task A           Task B
read  mem,reg
inc   reg
                 read  mem,reg
                 inc   reg
                 write reg,mem
write reg,mem
```

Now if we observed the memory location from the outside we would actually observe the sequence 0 -> 2 -> 1, with the final result being 1 (again). So the entropy of the system is physically present in the execution order of the concurrent instructions and logically expressed in the value found in the memory location. All that needs to be added now is a means to detect the race occurrence and pair this with a bit more significant probability of the race actually occurring.

adding detection

Single threads of execution can't detect execution time variations (if you maliciously intend to use a timer source like a TSC you lost because THAT is a second physically concurrent entity of execution...in my beautiful world of entropy controlled execution cheating is unheard of), so all we need to do to detect the race is *add a non-concurrent execution to execute concurrently with the concurrent execution.* Less confusing, take a thread of execution that only manipulates private variables and does not share any variables and do that concurrently with a thread of execution that does share resources. As the two threads do not conflict we can actually join them into a single thread. So we get

```
Task A           Task B
read  shmem,reg1
read  privmemA,reg2
inc   reg1
inc   reg2
                 read  shmem,reg1
                 read  privmemB,reg2
                 inc   reg1
                 inc   reg2
                 write reg1,shmem
                 write reg2,privmemB
write reg1,shmem
write reg2,privmemA
```

No matter what execution order we have if shmem,privmemA and privmemB all are initially 0, privmemA respectively privmemB is guaranteed to be 1 once both tasks have terminated. shmem though may be any of 1 or 2, thus the race detection is simply to check if shmem==privmemA+privmemB - in case it does not we hit a race condition. The second change - making it more likely to happen, is now simply to run this in a loop

```
thread A
    for(n=0;n<LARGE_NR:n++){
count++;
    }
thread B
    for(n=0;n<LARGE_NR:n++){
count++;
    }
...
 if(count != 2*LARGE_NR){
     /* race occured... */
```

Done - that's it - a simply entropy extractor for your PC, mobile phone, MCU in your ABS...

4.1.1 cnt.c

cnt.c is the simples TRNG I could think of, and yes - it is a TRUE random number generator - well it has some bias problems and its a bit load dependent but a part of the output actually does represent clean random bits. So its a bad TRNG but never the less the T is justified.

We need concurrency - so at least 2 threads, a shared object that needs to be declared volatile so the system actually is urged to store all intermediate values in main memory (how did people at IBM ever get the idea that volatiles are not use full for multi-threaded programs ? [**?**]) and finally to make the occurrence likely we use some large number 10 million should work on almost every system quite reliably (on the ABS controller of your car you might need more)

```
#define NUM_THREADS 2
volatile long cnt = 0;
unsigned long N=10000000;
```

Next we simple increment the shared counter without any protection around it so that we execute all permutations of the above read,increment,write interleavings possible.

```
void * Thread(void *v)
```

```
{
    unsigned long n;
    for(n=1;n<=N;++n){
        ++cnt;
    }
    return NULL;
}
```

Finally in the main routine we simply need to create the two threads and let them have their time to race. So after creating the threads simply block on joining them again.

```
int main(int argc, char **argv)
{
    int n;
    pthread_t t[NUM_THREADS];

    for(n=0;n<NUM_THREADS;++n){
        if(pthread_create(&t[n],NULL,
                Thread,NULL)){
            perror("pthread_create");
            exit(-1);
    }   }
    /* wait for completion */
    for(n=0;n<NUM_THREADS;++n){
        if(pthread_join(t[n],NULL)){
            perror("pthread_join");
            exit(-1);
    }   }
    printf("%d\n",cnt);
    return 0;
}
```

The only interesting line here is the final printf - which on my hot 1.6GHz AMD Duron (uniprocessor) gives me;

```
hofrat@rtl14$ gcc -O2 cnt.c -o cnt -lpthread
hofrat@rtl14$ ./cnt
15188241
hofrat@rtl14$ ./cnt
17009201
```

Just for completeness - this box is running a 2.4.26 kernel (Slackware 9.something) so this looks nice and random to some extent - how random is it ?

4.1.2 1st refinement gauss.c

To "prove" that this is really random what we are doing we start with a complicated way of producing normal distribution plots with a modern CPU by harvesting the inherent randomness. Not only are the curves a bit bumpy but you actually can't really say where they will be located - though for every box I tested on it really looks like this is a quite unique fingerprint of the hardware/software system. This code is striped of almost all error checking and all argument handling, showing the principle only.

The above code is not reprinted - so we only show the additional code here - we add a virtual urn to draw samples from. SAMPLES says how many balls to draw from the urn and NUM_TRIALS says how often to take SAMPLES from the urn.

```
unsigned long SAMPLES=256;
unsigned long NUM_TRIALS=256;
```

The thread code is identical to the one above. The urn now is the part that generates red and blue balls by letting the threads race and checking the result. We run this samples time to extract SAMPLES balls and determine the color of each ball - counting the red ones only. The color check simply is: race occurs ball it red else ball it blue.

```
int draw_balls(int *samples){
  int i,n;
  unsigned long red;
  pthread_t t[NUM_THREADS];
  red=0;

  for(i=0;i<*samples;i++){
    cnt=0;

    for(n=0;n<NUM_THREADS;++n){
        if(pthread_create(&t[n],NULL,
                Thread,NULL)){
            perror("pthread_create");
            exit(-1);
    }   }
    for(n=0;n<NUM_THREADS;++n){
        if(pthread_join(t[n],NULL)){
            perror("pthread_join");
            exit(-1);
    }   }

    if(N*NUM_THREADS - cnt)
        red++;
  }
  return red;
}
```

Main is not too exciting now it simply gets NUM_TRIALS sets of balls from the urn and print out how many were red and how many blue. Accu-

mulating this data and plotting it gives a more or less smooth Gauss curve.

```
int main(int argc, char **argv)
{
    int i;
    int num_red,num_blue;
    int sample_size=SAMPLES;

    i=0;
    while(i++<NUM_TRIALS){
        num_red=draw_balls(&sample_size);
        num_blue=sample_size-num_red;
        printf("%d %d\n",num_red,num_blue);
        fflush(stdout);
    }
    return 0;
}
```

So what is this doing ? Its letting a set of threads race on a unprotected global variable *cnt* - the run of the thread set is equivalent to drawing a ball from an urn of infinitely many read and blue balls. If a race is detected (cnt != N*NUM_THREADS) we assign this the color red, otherwise blue.

Here is what the Gauss curve on my humble AMD Duron looks like after 256 loops -

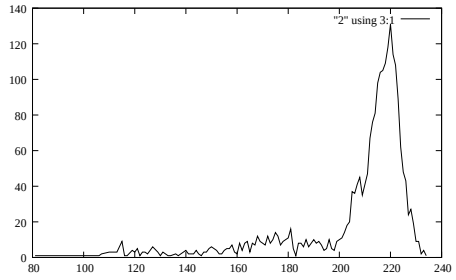

FIGURE 1: *gauss.c output with sample size 256*

ok - not much of a Gauss curve - but after running for about 3 weeks it would look nice and smooth... On a single core producing the Gauss curves is quite simple, it just takes a long time (a few weeks on an AMD Duron) - then you get something like:

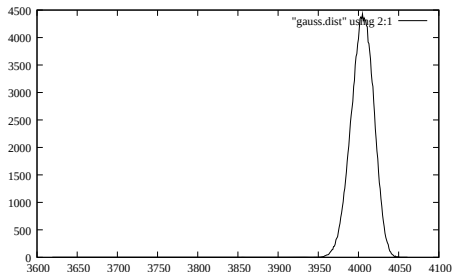

FIGURE 2: *gauss.c output with sample size 4000*

On multicore it starts getting a bit more involved as you no longer get a single curve but you get curve-sets - for 2 cores its still quite clear

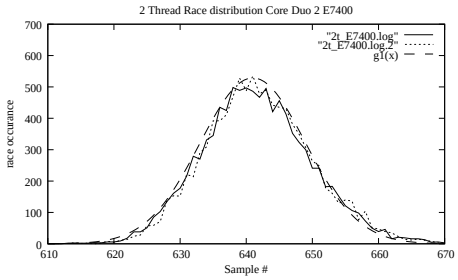

FIGURE 3: *gauss.c output on an Intel Quad Q6600*

with 8 cores its a bit messy - here is the result from running on a i7 (kernel 2.6.32 default Debian 6.0.3)

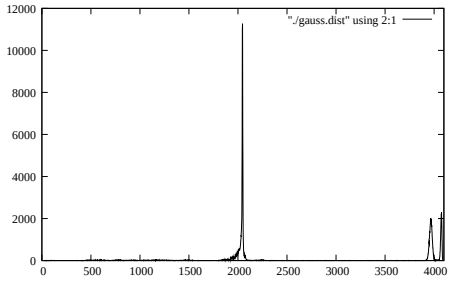

FIGURE 4: *multiple overlapping curves on an Intel i7 2620 CPU*

using gnuplot to curve fit individual peaks one can see quite nicely that they are produced by overlap of multiple normal distribution functions. The fitting was done in the one case against a single curve in the other case against 2 curves - might be possible to improve by using 3 curves - but the essence is visible in this form as well.

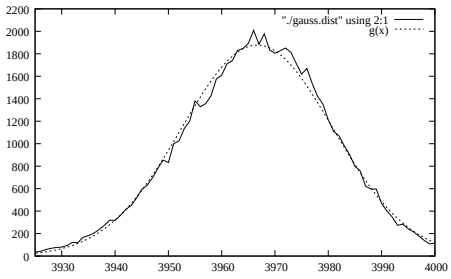

FIGURE 5: *detailed curve singled out from an i7 2620 CPU*

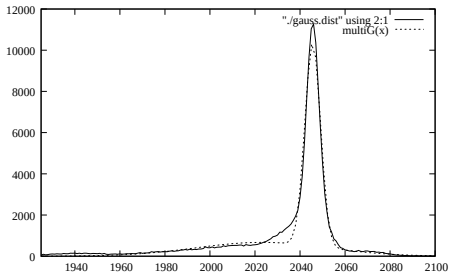

FIGURE 6: *curvefitting with gnuplot of the overlapping curves of an i7 2620 CPU*

The claim is that the occurrence of red/blue is truly random and as evidence this code produces close to perfect Gauss curves on and IDLE system. So this is not driven by any peripheral interrupts or the like, in fact on high loads things get a bit distorted... In my opinion this demonstrates that at the core of a modern CPU there is in fact a true entropy source at work - all we need to do is extract it.

And of course as this remarkable discovery was made by me, from now on, anytime you implement a race condition, it is Copyright Der Herr Hofrat ¡der.herr@hofr.at¿ 2009-2038 License GPL V2 [**?**].

running it

```
compile: gcc -O2 gauss.c -o gauss -lpthread
run: ./gauss | tee logfile
```

Be patient - this will take time if you want a smooth curve (hours..days...weeks !) once you have a suitably long recording plot the distribution - brute force might be

```
sort -n logfile | uniq -c > data
```

start up gnuplot and do:

```
gnuplot> plot "data" using 2:1 with lines
```

This will give you a nice Gauss curve on all systems (on some it just takes a while) - if it does not then your N needs adjusting ! Why ? because N determined the probability of a race occurring and that depends on the physical concurrency and how the hardware reads/writes memory - simply

- if the runs shows all samples close to 0 then increase N
- else if you get all samples close to SAMPLES decrease N
- else you should be fine

For those that see a feedback controller emerging right here - well that was esrng version 2 that actually used a control loop right at this point - but it turns out not to be quite that simple to build such a control loop - more on this later.

Back to the simple stuff, some values that work for me:

```
32 bit install:
AMD Duron UP          : 12000000
AMD Sempron UP        : 5000000
Core Duo E7400 SMP    : 100000
64 bit install:
Intel Nehalem 8 core : 10000
Intel CoreDuo 2 Quad : 5000
Intel i7              : 4000
```

If you do generate Gauss curves with this method, pleas do send me the N you used and a short system information - it would be interesting to see if the N that give 50

4.1.3 dist.c

gauss.c was used to get the distribution for a fixed loop value and establish a reasonable evidence of the underlying process actually being a stochastic - normal distributed process (not that this is a proof yet). The second dimension of interest is how the system behaves if one holds the outer loop (the number of trials and alters the inner loop, the loop counter of the entropy extraction (the race).

To take the step from playing with normal distributed curves to harvesting entropy effectively, one needs to now add a control loop. On some systems such a control loop seems to be quite trivial to establish - if we run the two thread race in a loop of lets say 10000 and do this with a loop length starting near 0 and incrementing in small steps - we get something like this:

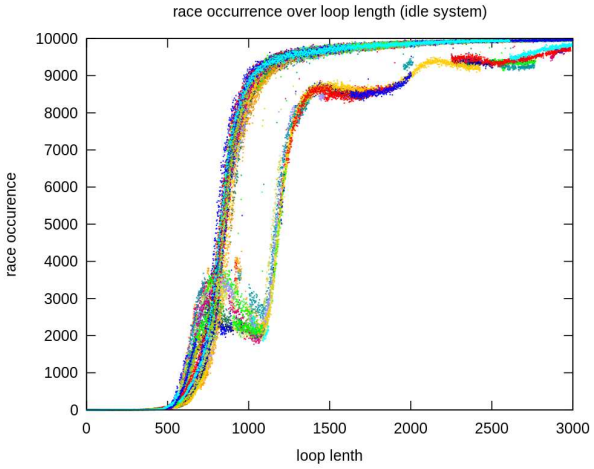

FIGURE 7: *30 runs of dist.c on a Intel Q6600 superimposed*

This is from a Intel Q6600 box (idle - default Debian 6 kernel 2.6.32). The distributions main branch looks quite usable for a set point at 50

So the control loop for the entropy harvester would seem quite simple at first, unfortunately the Q6600 distribution shown above is the good case ... *real* hardware looks a bit different. On a 32 Core AMD (thanks to the TU-Dresden OS folks !) it looks a bit wilder:

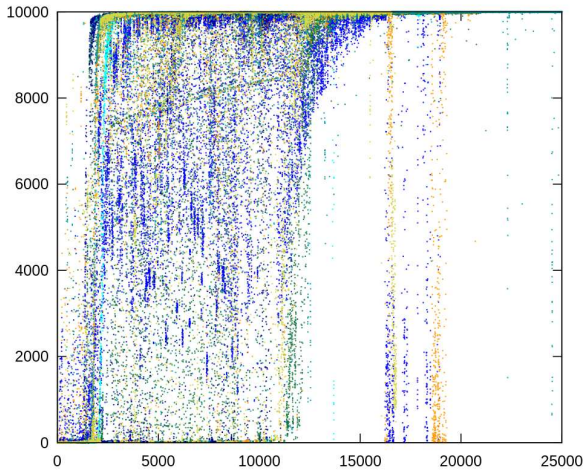

FIGURE 8: *8 runs of dist.c on an AMD Opteron 6276 superimposed*

This is only 8 plots superimposed - simply took too long to generate more - but the basic structure of the plot is visible - it is though no longer such a nicely visible functional connection of N and race occurrence probability. If you look a bit you will find some 1-exp curves in the plots noise. Finally here is the distribution of a i7 (idle - this time with a 3.3.4 kernel on Debian 6.0.3)

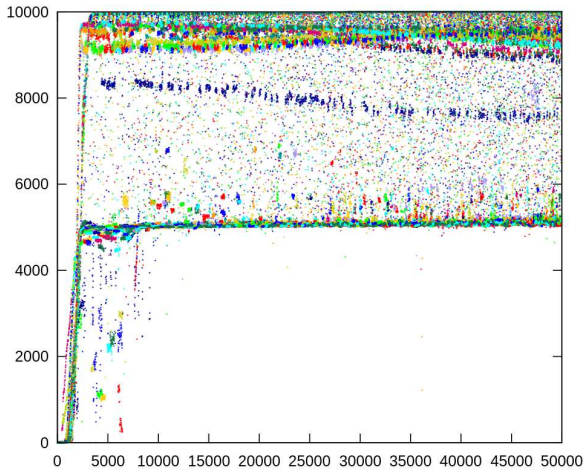

FIGURE 9: *30 runs of dist.c on an Intel i7 2620 superimposed*

and the i3 we played with did not look too different - the kernel version really has very little impact (if any) including the use of a RT kernel. Now given this distribution it should be clear why writing a controller for harvesting entropy is not that simple - the problem is that the system can be in a number of distinct states at any point in time and the occurrence of a race or no-race does not give you much information where you are in this type of plot - it well may though simply be that my meager understanding of control theory is at fault - if anyone has a hint on how to build a feedback controller for such a distribution, I would be interested in hearing of it - some ideas with Kalman filters are in the works.

4.2 esrng2.c

The simple feedback method did not work well (if anyone is interested in seeing the code - drop me a e-mail - its of course under GPL V2 as well). The alternative is to not use the distribution at all but use the single events and extend the concept from the ball-on-the-nail model to the Galton board. So next we look at the Galton board solution for the ESRNG.

A Galton board can be viewed as a ball-on-the-nail with state-recording added - so the location of the ball once it reached the bottom gives us some partial information about the intermediate states it had reached - we can't deduce the exact trajectory but we can say that if it ends up in a slot on the right

that it took the right path more often than the left path - and if it ends up all the way at the right we can fully deduce its exact trajectory - always taking the right path. So its a statistically state-preserving device in this sense.

Some might wonder why so much background is introduced first - the real reason is that embarrassingly simple random number generators are - well - embarrassingly simple. each of the two threads gets a unique prime number - to make things exiting I chose 2 and 3. Further we add a histogram that is used to record the intermediate values that occur in the inner loop. Note that this code is not protected against integer overflow in any way and the histogram is bounded by the histogram size. People familiar with C99 standards appendix J forgive me for the usage of undefined behavior. So for the *esrng2.c* the actual extraction loop simply is:

```
void * Thread(void *v)
{
    unsigned long long int n;
    unsigned int loc;
    int prime=(int *)v;

    for(n=0;n<N;n++){
        cnt*=prime;
        cnt/=prime;
        loc=cnt;
        hist[loc%HSIZE]++;
    }
    return NULL;
}
```

Four lines in the inner loop rather than just one - two of which extract the entropy. But what are those lines now doing ? First what we see here is a principle (well I believe it is a principle) if you want to harvest entropy your code must be perfectly deterministic for the single threaded case. If you run this with only one thread you would end up with the histogram having a peak of N at *prime* - that is 2 or 3 in our case here. With other words for the single threaded case this delivers a single peak in the histogram.

The problem with the above extraction loop (in cnt.c, gauss.c and dist.c) was simply that it was not state preserving so it was generating one bit per run basically. This code now is more or less like a statistically state preserving Galton board - every time you have a race you have a fair probability that you will enter a different "branch" of the Galton board and so improve the entropy extraction efficiency. Technically the difference is that the software version can stay at a particular location for some time - that is until a race occurs that switches the state.

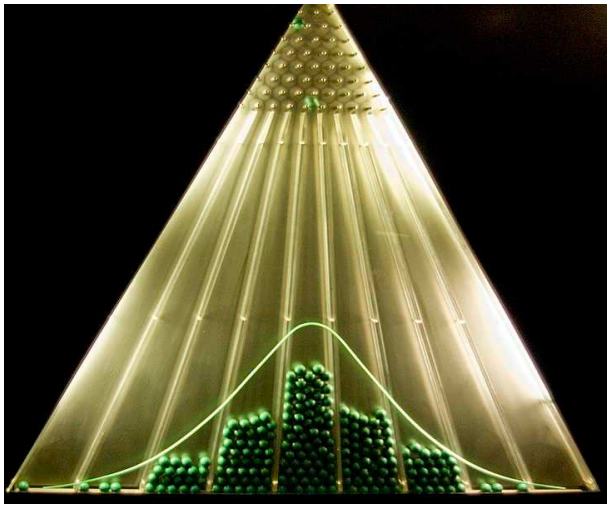

FIGURE 10: *Galton Board - This is a file from the Wikimedia Commons,* `http://en.wikipedia.org/wiki/File:Planche_de_Galton.jpg`

if at the end of the loop the cnt was multiplied by the threads prime then we call it a move left and if it was devised by the loops prime we call it a move right. The use of two different primes just increases the race detection probability - if you were to use the same value then some races would be mask. It also improves the histogram becoming dense - if both threads used 2 then it would of course be a sparse 2n histogram.

5 Results

Extracting random numbers from system entropy is a sensitive issue. For simulation or non-deterministic algorithms a biased TRNG might just lead to offsets or bad results - for security it could lead to very hard to detect vulnerabilities. Thus we are at this point not suggesting to use these methods for security related entropy harvesting. In some systems, like deeply embedded systems that have no alternative entropy source, it might be possible to use these methods. The premises being - if you have nothing better then a carefully tuned and monitored ESRNG *might* be better than a pure PRNG.

The threat is basically not at the algorithmic level but related to the parameters that need to be configured. At this point there are two configuration parameters in the ESRNG.

- N - the loop length used for the inner loop

- HSIZE - the size of the histogram used to record the intermediate values

If N is chosen too small - and as noted it is static (compile-time) in the current implementation - then quite obviously the probability of a race becomes quite small, thus the ESRNG would not fill the entire spectrum represented by the size of the histogram used for recording. One (weak) counter measure is to drop histogram values that are 0 at the end. If N is chosen larger than necessary, current tests results indicate that this will impact the bandwidth of the generated numbers only but not impact the quality of the ESRNG in principle. So N is relatively simple to chose and verify by testing (i.e. generate a 1GB sample and run the NIST test-suit on it - 1GB may take a few weeks...months to generate though).

HSIZE is a bit more tricky - if chosen too large or too small then the ESRNG will deliver "bad" random numbers in the sense that they are biased in some way. The HSIZE folds the overall spectrum generated into the range of 0...HSIZE and in this sense is compacting the spectrum of the ESRNG. The HSIZE it self does not contribute to entropy in any way - so there is no point in choosing HSIZE to be a prime number or the like (the same holds for N - following the rule of the single threaded model must be strictly deterministic). If HSIZE is chosen very large then the spectrum recorded will become sparse - even if 0 values are filtered the distribution of the numbers in the histogram which are the actual harvesting unit for entropy, are not homogeneously distributed over the range 0...HSIZE. Conversely if HSIZE is chosen two small then one will get patterns over multiple samples.

Note that the two values are actually not really independent of each other - on the test systems used the N value was actually always the same 800000000 and only the HSIZE has been adjusted to the system - the exception being the UP AMD Duron. Some tests also indicated that smaller values (and thus higher extraction rates) are possible. The coupling is non-linear so if N is very much larger than needed it can not be compensated with increasing HSIZE.

From these observations the tests were derived. Testing was done on multiple platforms from single core x86 and ARM processors, dual core X86 and ARM and 4,6,8,16 and 32 core X86. Unfortunately no PowerPC available for testing this stuff. Tests were run on some of the test-systems with the box completely idle (text-mode, no network connected, other than the default system processes and the shell used to launch the tests no processes running) , where multiple systems of the same type were available the second one would run with varying load settings. Other systems where guest accounts were available ran the ESRNG tests with a nice value of 35 and were simply exposed to what ever the load was on those systems.

In all of the test-systems two sets of tests were run, the one is a continuous harvesting as a "entropy daemon" - basically just running in a tight loop, as soon as a sample was drawn the loop was restarted, and the second was harvesting a block of bytes and concatenating them together. The block size (HSIZE) varying from 256 bytes (AMD Duron 1.2GHz) to 4kB (i3,i7,Q6600). The background of these two tests are that for this RNG two modes of use are envisioned

- Demand Mode: generate N bytes of random data on demand by launching an ESRNG instance.
- Daemon Mode: continuously run the ESRNG and buffer data in some fifo for later use.

Both modes work - once HSISE and N have been set properly. The main difference is that the bandwidth in daemon mode is a bit lower, simply because it can not be run with a high nice value. This is because it would produce an irritatingly high CPU load due to the permanently active threads - so the daemon mode was running with a nice value of 35 or 39 (this was of course also to proof that Tannenbaum is wrong with his claim that nobody ever uses the nice capabilities of UNIX [2]). Also the test was significant to ensure that the good results from sampling 2k or 4k chunks is not dominated by some sort of start-up effect (COW induced page-faults, parent/child timeslice handling etc.)

The bandwidth varies between different systems, the AMD Duron 1.2 GHz delivered about 2.5k per hour (compared to /dev/random this still qualifies as hot !), to about 200 bytes per second on an i7 (i7-2620M 2.7GHz). The actual generation rate for a single chunk of course is random.

5.1 evaluation of results

The preliminary evaluation of results was done with two quite simple methods. The first is to continuously check each of the emitted samples (roughly HSIZE each) with the random.org tool *ent*. The second was to generate samples of equal size using /dev/urandom and comparing the quality of the two sources again with the random.org tools. Ultimately the goal is to run the NIST test-suit on samples -

but at this point we simply don't have the necessary 1GB samples (except for a single system).

The results shown here as plots are only from a few of the systems. The random.org test suit uses entropy, mean, chi-square, montecarlo Pi and serial correlation as measures for the independence of samples. A test-run was considered clean if it:

- did not show a permanent bias on the mean and pi
- entropy and serial correlation converged towards 8.0 respectively 0
- Chi-square stayed in the 5 to 95 % range - peaks of course were permitted.

the measures were done for the individual samples and for the concatenated samples. Generally the ESRNG would show better values than /dev/urandom - notably for large samples - for smaller sample sizes it is hard to say much. A comparison with /dev/random is unfortunately not feasible as it is not possible to generate megabyte size samples from /dev/random - the small samples that are feasible are not conclusive.

The mean values all approach 127.5 eventually with some oscillation around this value - the amplitude of the oscillation diminishes with increased sample sizes and shows very stable behavior. Montecarlo Pi is a bit of a headache for me as I have at least two systems that are not approaching 3.141596 but rather reach a quit stable value of 3.142X - both of these systems are Intel Q6600s, which of course is not a satisfactory explanation in any way - currently they are at about 250MB - once the 1GB threshold is reached data will be subjected to the NIST test-suit and we hope this will clarify if the data is actually problematic or if this is an artifact of the random.org test-suit.

Entropy on all systems shows a very rapid increase and clearly performs an asymptotic approach towards 8.0 - 8.0 has been reached with samples sizes of more than 300MB (although only two systems have been running long enough to reach this) - we expect other systems also to reach this level eventually. Similarly with serial correlation that approaches 0 (roughly with sample sizes in the 100MB range) and then oscillates with a very low amplitude around 0.

The most sensitive of the tests really is the chi-square test - though in all cases where the chi-square went bad (that is approached 0 or 100 % probability of randomly exceeding the chi-square value of the sample) the other parameters (except for Entropy) would eventually follow. Chi-square though can not be used as a hard filter - samples have hit close to 0 or 100 % and then rebound to a stable range between 10 and 90 % - basically it is expected that for a large set of samples the distribution would include the entire spectrum - and this clearly is the case for the current *esrng2.c* code base.

5.1.1 Preliminary Data

Run-time env is a Debian 6.0.3 with the default 2.6.32 kernel, 64bit installation - default desktop environment (so basically what you get when you hit enter as often as possible during installation). The system was used during the test for normal office work - load average was generally very close to 2 (which is what is induced by *esrng2* running). *esrng2* was running with a nice value of 35.

The data samples generated are still too small for conclusive judgment as noted above. The plots here are a snapshot of the currently running tests on different platforms. The essential point being that all of the screening tests converge nicely on all platforms and chi square is nicely covering the entire range as it should be if the samples are truly random.

The first set of plots is a comparison of entropy, montecarlo pi, serial correlation and mean value for three systems (Core i7, i3 and PentiumD)

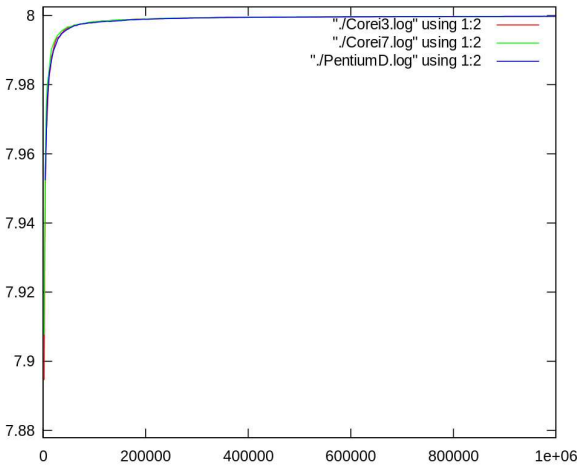

FIGURE 11: *Entropy test results on i3,i7 and PentiumD*

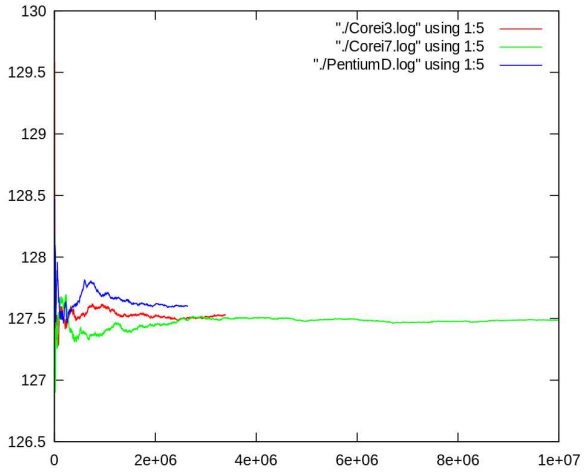

FIGURE 12: *Arithmetic average of random data on i3,i7 and PentiumD*

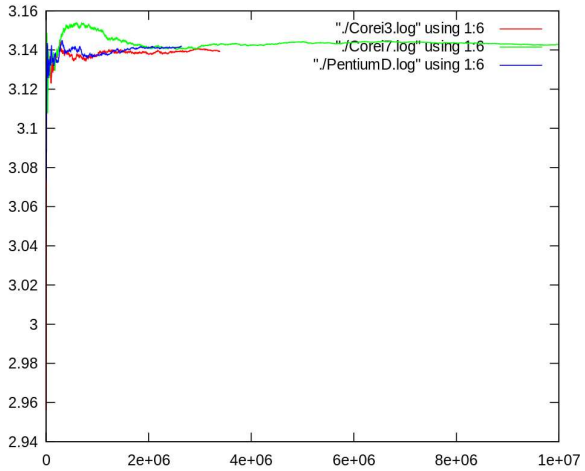

FIGURE 13: *Montecarlo Pi test results on i3,i7 and PentiumD*

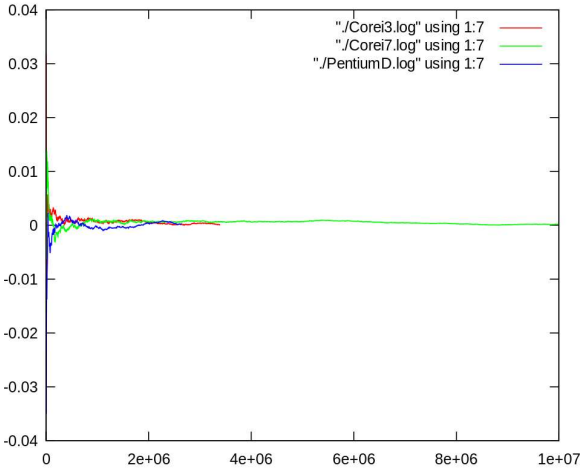

FIGURE 14: *Serial correlation tests results on 3 systems*

As is clearly visible they all converge - the i3 was more or less totally idle , the i7 was used as normal desktop system while running the tests. The PentiumD was only sporadically used.

The second batch are the chi square tests run on the cumulative data as it is generated and as would be expected from a truly random source they do not converge or stabilize in any way.

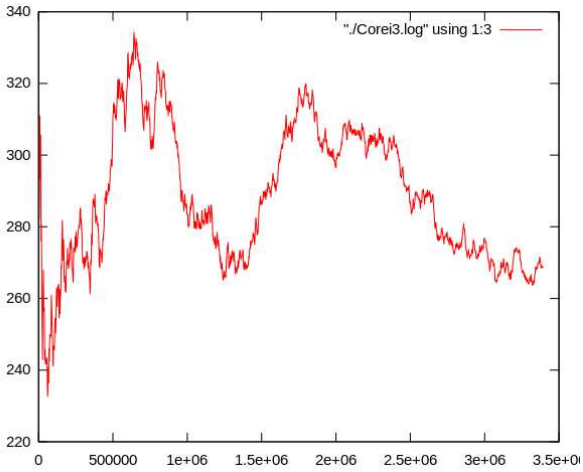

FIGURE 15: *Chi square on the i3*

FIGURE 16: *Chi square on the i7*

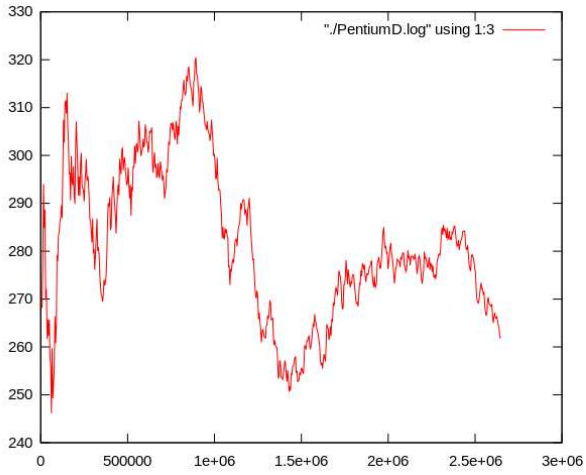

FIGURE 17: *Chi square on the Pentium D*

Tests have been running on all together 10 different systems to date and on all of these systems stable settings for HSIZE could be found with a relatively small effort. This is no proof of reliable random number generation yet, but it is a indicator that continuing this work makes sense.

5.2 Next Steps

The most important next step is to gather sufficient 1GB samples from different systems (hardware, kernel version, load-scenarios) and run the NIST testsuit on these samples - this will take some time though.

Further completing the test-systems used is going to be an important step - the systems used at this point are only a small subset of architectures available and it is too early to claim this method is generalizable. Notable the lack of PowerPC and MIPS platforms is problematic.

Some first runs on KVM and XEN guests have been done - though the runs are not yet long enough to say much - one problem that did show up though was that migration is a problem as it mandates recalibration or rather switching of parameters to fit the new environment. The currently used static settings seem to not be reliable on guest OS - but more work on this is needed.

Finally extending this to other operating systems is on my TODO list - notably running it on L4 type microkernels as a server would be very interesting - while a first quick shot was done, no systematic work has yet been done on this and no reliable (even speculative) results have been achieved.

With respect to the *esrng2.c* code it self, the focus is on getting a better understanding of the entropy sources and developing a suitable adaptive control algorithm so as to allow an increase of the entropy harvesting. Some experimental adaptive code *esrng1.c* that unfortunately ended up having 9 parameters has show at least an order of magnitude higher entropy extraction rates on some of the test-systems than the current implementation, but we have been unable to automate calibration rendering this approach infeasible for practical use at this point.

A side-issue is the use of the distributions (dist.c and gauss.c) to "finger-print" systems - some superficial checking has been done - and at least the systems I currently have access to can all be identified by their fingerprints. This well may be a useful fallout from this work to investigate in more depth.

6 Conclusion

This work is an empirical approach to the problem based on a very simple initial model - to harvest physical non-determinism we ideally need a deterministic system. The main claim of this paper is that to produce good random numbers based on the inherent physical randomness of modern superscalar CPU, most notably multicore systems, we need a deterministic system to serve as an entropy extractor. The hardware/software system is of course not deterministic so the use a feedback controller to mitigate the shortcomings of the real systems was proposed but problems with calibration prevented this from yielding practically useful results. A less aggressive solution by using statistical state preservation and recording the final states in a histogram shows reasonable bandwidth compared to /dev/random at least, and simple (static) configuration.

You might ask - who needs this ? We have */dev/random* for those being strict, */dev/urandom* for the impatient and for the hard-core guys we have TRNGs at any price category you can imagine. The majority of computer systems in this world are embedded systems - many deeply embedded and only exposed to irritatingly deterministic interrupt sources - thus really bad entropy sources like a CAN bus or a redundant 1553. At the same time these systems have discovered that security is an issue never the less (see EN 50159 Ed 2 2011). For these systems we envision that extracting the entropy from our truly - and physically - non-deterministic CPUs that we are using would be more than just a nice thing to have. With other words we all have a TRNG

in the system - the CPU - all we need is an entropy extraction utility to harvest it.

The implementation is based on utilizing a well established technology - the race condition - while there are elaborate implementations of the same we have chosen to base this work on a very conservative race condition - a single unprotected integer. Effectively any reasonably complex software construct thus would offer almost infinite offerings for entropy extraction - any point where you use protection in the system is a potential source of entropy - if it were actually predictable from the access pattern why would you be using a lock ? What we need to do is understand better how to extract entropy from the vast source modern super-scalar CPUs offer and integrate the entropy extraction in the primitives used to enforce deterministic access to shared objects.

The actual implementation is a proof-of-concept and no more - it demonstrates the ability to extract entropy with an embarrassingly trivial piece of code - a two line race condition.

The big unanswered question though is - does this code not violate copyright ? there is well known prior art on implementing race conditions in operating systems probably going back to CTSS - we simply hope nobody will notice.

References

[1] *General Public License Version 2*, http://www.gnu.org/copyleft/gpl.html

[2] *Modern Operating Systems*, A. Tannenbaum, 2007

[3] *nalysis of inherent randomness of the Linux kernel* Nicholas Mc Guire, DSLab, SISE, Lanzhou University, Gansu, China, Peter Odhiambo Okech, Faculty of Information Technology, Strathmore University, Nairobi Kenya. , 2009, RTLWS11

[4] LibMTPRNG: A Multithreaded Pseudo Random Number Generator Dr. Dobb's Journal, 2009

[5] *http://docs.python.org/library/random.html*

[6] *CNT Device is Basis for Random Number Generation*, `http://www.nanotech-now.com/news.cgi?story_id=11658`,2011

[7] *RAND Corporation*, 1955

[8] *Introduction to Probability* , Charles M. Grinstead, J. Laurie Snell, Dartmouth Colleg , 2011,