INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

HArdware Volatile Entropy Gathering and Expansion: generating unpredictable random number at user level

André Seznec Nicolas Sendrier*

Thème 1 — Réseaux et systèmes
Projet CAPS

Publication interne n ° 1492 — Octobre 2002 — 20 pages

Abstract: The availability of a random number generator with high cryptographic qualities on a computer is one of the central issues of cryptographic implementations.

HAVEGE (HArdware Volatile Entropy Gathering and Expansion) is a new software heuristic for generating unpredictable random numbers on PCs and workstations. PCs and workstations are built around modern superscalar microprocessors. These processors feature complex hardware mechanisms that aim to increase performance. A significant part of the global state of the microprocessor is not architecturally visible through the instruction set (e.g. caches, branch predictors and buffers). HAVEGE leverages the uncertainty introduced in the internal states of the processor by external events. HAVEGE combines entropy/uncertainty gathering from the architecturally invisible states of a modern superscalar microprocessor with pseudo-random number generation.

First we show that the hardware clock cycle counter of the processor can be used to gather part of the uncertainty introduced by operating system interruptions in the internal state of the processor. Tens of thousands of unpredictable bits can be gathered per operating system interruption in average. Then, we show how this entropy gathering technique can be combined with pseudo-random number generation in HAVEGE. Since the internal state of HAVEGE includes thousands of internal volatile hardware states, HAVEGE features a very high security level. HAVEGE also reaches an unprecedented throughput for a software unpredictable random number generator: more than 100 Mbits/s with off-the-shelf workstations and PCs.

Key-words: Superscalar processor, cryptography, hardware counters, unpredictable random number generators, operating system, operating system interruption.

(Résumé : tsvp)

* CODE project, INRIA Rocquencourt

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

HAVEGE: un générateur logiciel d'aléa imprévisible en mode usager

Résumé : Nous présentons HAVEGE, un nouveau générateur de nombres aléatoires imprévisibles. HAVEGE exploite l'interaction des états matériels non architecturaux des processeurs superscalaires modernes et des systèmes d'exploitation.

1 Introduction

We define an *unpredictable* random number generator as a practical approximation of a truly random number generator. Instead of formally proven uniform distribution and formally proven irreproducibility, an unpredictable random number generator features *practical uniform distribution* and *practical irreproducibility* which are defined as follows:

- **Practical uniform distribution**: it is computationnally unfeasible to distinguish between the distribution of the outputs from the generator and the uniform distribution.

- **Practical irreproducibility**: for any observer or even the user itself, there is no practical way to exactly complete an uncomplete sequence of generated numbers.

The availability of an unpredictable random number generator on a computer is one of the central issues of cryptographic implementations. Both current hardware and software unpredictable random number generators available with off-the-shelf PCs and workstations are quite slow (respectively 10-100 Kbit/s [11] and 10-100 bit/s [9]). Due to their limited throughput, these generators are generally used for generating seeds for software pseudo-random number generators.

HAVEGE (HArdware Volatile Entropy Gathering and Expansion)[1] is a new software heuristic for generating unpredictable random numbers on PCs and workstations. PCs and workstations are built around modern superscalar microprocessors. These processors feature complex hardware mechanisms that aim to increase performance. A significant part of the global state of the microprocessor is not architecturally visible through the instruction set (e.g. caches, branch predictors and buffers). HAVEGE leverages the uncertainty introduced in the internal states of the processor by external events. HAVEGE combines entropy/uncertainty gathering from the architecturally invisible states of a modern superscalar microprocessor with pseudo-random number generation.

First, we present the HAVEG (HArdware Volatile Entropy Gathering) family of algorithms. These algorithms indirectly gather (part of) the uncertainty introduced by external events in the internal volatile hardware states of the processor from the memory hierarchy and the branch prediction mechanism. The HAVEG algorithm relies on the hardware clock cycle counter. This counter is used as an indirect probe to extract (part of) the entropy from the internal volatile hardware states. On current PCs and workstations, the HAVEG algorithms generates several tens of thousands of unpredictable random bits, in average, per every operating system interruption.

Then, we extend the HAVEG algorithms to the HAVEGE generator (HArdware Volatile Entropy Gathering and Expansion). HAVEGE combines HAVEG-like entropy gathering with simple pseudo-random number generation (walk in a self-modifying table). The HAVEG-E generator combines two major qualities; a very high throughput (more than 100 Mbits/s

[1] We coined this name in respect with entropy gathering techniques since we exploit the same sources (i.e. external events) of uncertainty in the execution of a process.

of unpredictable random numbers) and a very high security level. The security of HAVEGE relies on its internal state. This internal state consists of classical data mapped in memory and in thousands of binary volatile hardware states. The global status of those hardware states is not accessible, even for the user running the generator. Any external event introduces major perturbation in this internal state of the generator. Any attempt to indirectly collect the invisible part of the internal state alters it.

The remainder of the paper is organized as follows. Section 2 briefly discusses the current practice on unpredictable random number generators. In Section 3, we describe some of the states of internal components that may generate uncertainty in the execution time of simple sequences of instructions. Then we point out that most of these states are not directly monitorable by the user and will also be modified by all the external events. Section 4 presents quantitative measures of the non-architectural hardware states that are modified by operating system interruptions in the memory hierarchy and in the branch prediction tables with a Sun Solaris workstation featuring an UltraSparc II and with a Linux Pentium III workstation. Section 5 introduces the HAVEG family of algorithms. These algorithms gather the uncertainty introduced in internal volatile hardware states by operating system interruptions. Section 6 presents the HAVEGE generator: a user level self-modifying random walk (the walk is self-modified using the hardware cycle counter) is used to both generate sequences of unpredictable random at an unprecedented rate (up to 100 Mbit/s on current processors) and continuously gather and propagate the uncertainty introduced by external events in internal hardware states. Uniform distribution qualities are analyzed. The internal invisible state of the HAVEGE generator is analyzed. Finally, Section 7 summarizes this study.

2 Random numbers in cryptography

Random numbers are used in many cryptographic applications like secret key generation or authentication protocols. If an opponent can guess even a small proportion of the random bits, then the security of the system may be dramatically endangered. Being able to produce safe random numbers is a necessary condition for implementing cryptography.

2.1 Practice

Hardware truly random number generators rely on non deterministic processes. The most convincing implementations use quantum mechanics [10, 19] and can achieve a rate of up to one megabit per second. Radioactivity can also produce truly random bits [21] but at a much lower rate (240 bits per second). Many other hardware generators exist. They usually exploit the thermal noise in electronic devices. For instance, Intel Random Number Generator [17, 5] uses such a noise, and performs at an average rate of 75 kilobits per second [11]. However, while these generators are based on non deterministic physical process, most of them are slightly biased and do not pass uniform distribution tests [9] (Section 4.1 in [9]).

In the absence of dedicated hardware random number generators on a computers, there is a possibility to exploit the randomness from chaotic air turbulence in disk drives [2].

However Jakobsson et al. [9] evaluated to only 5 random bits per minute, the amount of "proven" randomness that could be extracted by this process. Such an approach is therefore not very practical. Jakobsson et al also proposed a "utility" mode where 577 bits per minute are obtained. Standard statistical tests [15] were unable to distinguish the output of the utility mode from truly random sequence.

Fortunately, many events occurring in a computer, though deterministic by nature, are sources of uncertainty. RFC 1750 [8] explains how the occurrences of various events (mouse, keyboard, disk, network, ...) can be used to build a random number generator. Such generators have been implemented; first in Linux (`/dev/random`), then for most Unix platforms with the Entropy Gathering Daemon[2]. However, they perform at a relatively low rate (a few bytes per second for `/dev/random` if the machine is not active). Practical solutions in cryptography use a pseudo-random number generator whose seed is obtained by entropy gathering techniques. This is the case, for instance, of the random number generator of PGP[3] and of Yarrow [12][4].

Our approach is an extension of these entropy gathering techniques. These implementations were using only measurable external parameters (date, duration, size of the data, data itself, ...). Our approach leverages the modifications that external events induce on the global state of the machine particularly on unmonitorable internal states.

2.2 "Demonstrating" unpredictable randomness

Unpredictable random generators have to exhibit two important qualities. First the distribution of the generated sequence of numbers must be as uniform as possible. Second, reproducing the exact sequence of generated numbers should be impossible in practice.

2.2.1 "Evidence" of practical uniform distribution

Proving that a sequence of bits is uniformly distributed is virtually impossible. However most non-uniform sequences have properties that make them fail some algorithmic tests.

The battery of tests for uniform distribution used in this paper includes the FIPS-140-1 test sequence for random number generators[5], entropy test, chi2 test, Monte Carlo tests [7, 14], [15][6] and the NIST statiscal suite[6]. Diehard was also used as the empirical evidence of the randomness of their "utility" mode random generator by Jakobson et al. [9].

Failing these tests will prove that a sequence is not uniformly distributed. Passing the tests does not prove that the sequence is completely random, but might be considered as a good indicator that finding (and exploiting) a bias in the sequence will be an unsurmountable task.

[2] `http://www.lothar.com/tech/crypto/`

[3] Pretty Good Privacy, `http://www.pgpI.org/` or `http://www.pgp.com/`

[4] `http://www.counterpane.com/yarrow.html`

[5] see `http://csrc.ncsl.nist.gov/fips/fips1401.htm`

[6] DieHard is a popular battery of tests for uniform distribution of number sequence that has been developed by George Marsiglia over the last thirty years. DieHard is available at `http://stat.fsu.edu/~geo/`

2.2.2 Showing practical irreproducibility

In order to show the practical irreproducibility of the generated sequences, we will show that there is no mean for the user itself and a fortiori for an external observer to collect the internal state of the generator.

2.3 Security assumptions

Unpredictable random number generation is seen as a matter of security. Security involves many components that may be individuallt attacked. Because the purpose of this paper is only unpredictable random number generation, we will suppose that the opponent may access the output of the random number generator, that he/she may log on as a user and even as a superuser to similar hardware platforms where the random number generator is running. We even consider that he/she has access to the platform (e.g. time sharing) where the random number generator is running. However we assume that he/she can run only in user mode, for instance he/she can not read or write data sections belonging to the random number generator.

3 Parameters influencing the execution time of a sequence of instructions

In microprocessor systems built around modern superscalar processors, the precise number of cycles needed to execute a (very short) sequence of instructions depends on many internal states of hardware components inside the microprocessor as well as on externalevents to the process.

In this section, we first list some of the components and associated internal hardware states of modern superscalar processors that impact the number of cycles that separates two successive reads of the hardware cycle counter. Then, we discuss the means that are available for an attacker to reproduce (guess with a reasonable probability of success) these internal states of the processor.

3.1 Components

Let us consider a simple sequence of instructions including at least one conditional branch and one load/store that separates two consecutive reads of the hardware clock cycle counters.

The number of cycles for executing this short sequence will depend on branch prediction outcome and on the presence or absence of data and instructions in the memory hierarchy (L1 and L2 caches, TLBs). In addition to these binary status (present/absent or correct/wrong), the execution time of a sequence also depends on the precise status of all instructions in all stages of the execution pipeline. Moreover modern superscalar processors feature numerous buffers that optimizes performance. The response time of the memory hierarchy servicing a miss depends on the precise status of all these buffers. Additionnaly, the response time of the memory on a L2 cache miss will depend on any conflicting event on the memory system or on the system bus.

3.2 Reproducibility of the internal states of the processor

Our security assumption (Section 2.3) is that the attacker can access the platform where the random generator is running in user mode, but not as a superuser. Using the hardware cycle counter in order to produce unpredictable random numbers would not be a valid approach if this attacker was able to reproduce the global state of the processor and then to reproduce the sequence.

PC or workstation instruction sets do not provide user level direct ways to monitor the internal states of the processor. The user will be able to indirectly detect the presence or the absence of a data block in the cache by using the hardware clock counter in conjunction with the read of the data. Such an action will displace other data blocks and instruction blocks and will also modify buffer states and branch prediction table entries. The knowledge of the global state of a microprocessor for an arbitrary point in any program is unattainable for the proprietary of the process and *a fortiori* for the attacker.

The only possible mean for an observer to record the global internal states of a microprocessor system would be to freeze the hardware clock and probe all states in the processor. Such functionality is provided for testing the processor in development labs, but is not available at user level on commodity PCs and workstations !

The number of cycles needed to execute an instruction sequence does not depend on the whole global state of the processor, but only on the fraction of the hardware that it really activates. For instance, a loop which task only consists of reading the hardware clock counter will exhibit a very regular behavior. An attacker would have a reasonable probability to reproduce a significant part of the same sequence. Therefore the random generator should activate a large fraction of the unmonitorable hardware states in the processor.

However, no matter the activated hardware complexity, in the absence of external injection of new states, any algorithm would only produce deterministic results. The operating system interruptions (and all the external events) introduce modifications of the internal hardware states of the processor.

As a consequence:

> *A robust unpredictable random generator should activate a significant amount of unmonitorable hardware states that are touched with each operating system interrupt.*

4 Operating system interruptions and unmonitorable hardware states

Among the states that are affected by an operating system interruptions, let us cite the contents of the instruction and data caches, the translation buffers (TLBs), the L2 cache and the branch prediction structures.

In this section, we report estimates on the minimum numbers of blocks or entries that are displaced from data/instruction L1/L2 caches and instruction/data TLBs by a single operating system interruption. Intuitively this represents a minimal evaluation of the perturbation introduced by the interruption. We also report the "minimum" cumulated perturbation with 100 consecutive interruptions. These numbers are reported for a non-loaded machine (no other heavy process running) since with a loaded machine more blocks (in average) will be evicted.

Our evaluation was done on a Sun Workstation featuring an Ultrasparc II running with Solaris. A partial analysis of a Linux PC featuring a Pentium III is also presented.

4.1 Experimental methodology

We consider here five memorization structures. We successively measure the impact of process interruptions by the operating system on these hardware structures as follow:

An algorithm using a working set that meets the size of the hardware structure is ran. A while loop on the hardware clock counter waits for the occurrence of an operating system interruption. On the return from the interruption, while the working set is reinstalled in the hardware structure, the hardware clock counter is used to determine which element were displaced by the interruption. When accessing back a data/instruction block (or a new page) after the return from the interruption, we timed its access with the hardware counter. For each of the structures, we were able to determine a threshold indicating whether the block has been displaced or not for the UltraSparc II. For the Pentium III featuring out-of-order execution, our evaluation was much more difficult and is only partial.

4.2 Analysis on UltraSparc II and Solaris

4.2.1 L1 data caches

The UltraSparc L1 data cache is 16Kbyte and direct-mapped. It features 512 32-byte cache sectors. A miss fetches only 16 bytes while the second 16-byte block will be fetched only on demand. The state of sector location is therefore represented by the physical address of the data sector mapped onto it and the presence/absence of the two halves of the sector.

We experimentally measured that on a non-loaded machine most of the operating system call touch about 80-200 data cache sectors (with a peak around 100-110 cache sectors) while, depending on the runs, 1-10 % of the operating system calls displace almost all the blocks from the cache. For 100 consecutive interruptions, the number of displaced blocks always exceeded 11,500 in our experiences.

4.2.2 L1 instruction cache and the conditional branch predictor

The 16Kbyte instruction cache on the UltraSparc is 2-way set-associative and features 32-byte cache blocks. On the UltraSparc, the branch predictor is incorporated in the I-cache: a 2-bit counter is associated with every pair of instructions and a prediction of the address of the next 4-instruction block is associated with every 4-instruction group.

The state of a cache set can be represented by the ordered set of the instruction blocks addresses mapped onto it and the associated branch prediction information. An operating system call will flush down part of the I-cache, and therefore will also flush part of the branch prediction information.

We measured that, on a non loaded UltraSparc machine, most operating system interruption displace around 250 32-byte blocks of instructions , while 100 consecutive operating systems displace at least 30,000 blocks[7].

4.2.3 TLBs

The UltraSparc II features a data TLB and an instruction TLB. Both TLBs have 64 entries and are fully associative and feature a Not Last Used replacement policy. The global state of the TLB can be represented by the set of the addresses of the pages mapped by the TLB and the state of the logic needed for implementing the replacement policy.

We experimentally measured that, on a non loaded machine, every operating system interruption displaces a significant amount of data TLB entries (minimum 16, 52 in average !), but only displaces a few instruction TLB entries (6 in average). For 100 consecutive operating system interruptions, the minimum cumulated number of displaced blocks always exceeded 4,500 for the data TLB, but only 600 for the instruction TLB.

4.2.4 L2 caches

The UltraSparc II processor is used in conjunction with a 1 Mbyte L2 cache featuring 64-byte blocks.

In the vast majority of cases, an operating system interruption replaces between 850 and 950 blocks. The minimum cumulated number of ejected blocks for 100 operating systems interruptions always exceeded 95,000.

4.2.5 Summary on UltraSparc II and Solaris

On a Sun workstation featuring an UltraSparc II and Solaris, the five memorization structures we have considered in this section are subject to lose a significant amount of volatile non-architectural hardware information on operating system interruptions.

4.3 Partial analysis on Linux Pentium III PC

While the internal hardware states in an out-of-order execution processor exhibit more opportunities for uncertainty that an in-order execution processor, the quantitative analysis of the hardware states modified by operating system interruptions on a Pentium III (or any out-of-order execution processor) is much more difficult than the one presented before.

For instance, due to out-of-order execution, it is very difficult to discriminate between a miss and a hit in a cache (partcularly the instruction cache), just through the number of cycles between two successive reads of the hardware counter.

However we were able to measure that about 150 32-byte blocks in average are replaced from the L1 cache per OS interruption, and, that, opposite to Solaris, the distribution of the

[7] This measure is particularly difficult to get precisely. In order to get it, we had to build a basic block of 4096 instructions that is executed in 1024 cycles (as close as possible) when the complete sequence hits in the cache. Then we measure the execution time of the sequence after the operating system interruption and get an approximation of the number of misses by dividing the execution time minus 1024 cycles by 7 (the minimum miss penalty on the UltraSparc). The assumption that the miss is served in a minimum delay holds because our code did not exercise at all the data cache.

number of evicted blocks had two major points around 10-20 blocks and between 200-260 blocks.

We also showed that every operating interruption displaces at least 40 entries from the BTB and that many of them displace more than 100 entries. Note that on the Pentium III, replacing an entry in the 512-entry BTB is displacing a branch target address, but also 36 bits for predicting branch direction.

4.4 Summary

While the numbers presented in this section are only valid for the considered platforms, the same conclusion will prevail for other processors and other operating systems for PCs and workstations. For instance, other processors (e.g Alpha 21264, Pentium 4) features more complex branch prediction mechanisms that will be more affected by the operating system than the ones on the UltraSparc II or Pentium III.

5 HAVEG algorithms: gathering (part of) the uncertainty injected by operating system interrupts

A large number of internal unmonitorable hardware states are modified by any operating system interruption. Therefore one can expect that a significant uncertainty is injected in these volatile hardware states on each operating system interruption.

We present a simple family of entropy gathering algorithms, the HAVEG (HArdware Volatile Entropy Gathering) algorithms. Unlike previous entropy gathering implementations, HAVEG algorithms are ran at user level. The HAVEG algorithms use the hardware clock counter to gather uncertainty from a short sequence of instructions that touches a few unmonitorable hardware states. We have designed HAVEG algorithms to exercise respectively the data cache, (part of) the instruction cache and the branch predictor, the data TLB and the L2 cache.

The HAVEG algorithm we use for gathering uncertainty from the instruction cache and branch prediction structure is detailed below. Then we present the volumes of unpredictable random numbers that can be extracted from different processor components.

5.1 Gathering uncertainty from the I-cache and branch prediction tables

Figure 1 illustrates the HAVEG algorithm we used for gathering part of the uncertainty injected in the instruction cache and branch prediction tables.

HARDTICK() is a function that reads the hardware clock counter. It also tests the difference with the last read value. The NBINTERRUPT counter is incremented whenever this difference is higher than a threshold indicating that there was an interruption between two successive calls to the function reads.

Throughout the loop, *HARDTICK()* is called many times and the result of this read is combined through exclusive-OR and shifts in the *Entrop* array. The unrolling factor (XX) was adjusted for each of the targeted architectures. The compiled loop body just fits in the instruction cache (the trace cache for the Pentium 4) and does not overflow the branch prediction structures.

The while loop is run until the number of encountered interruptions reaches NMININT. SIZEENTROPY is the size of the table used for gathering the read value of the hardware clock counter.

Note that the flow of instructions executed by the loop body of the algorithm is completely deterministic. Therefore, in the absence of operating system interrupt, the content of the instruction cache and also, of the branch predictor should be completely predictable: we checked that the iterations of the loop just after an interruption present much more uncertainty that the iterations that occur long after the last iteration.

```
int Entrop[SIZEENTROPY];
int A;
 1   while (INTERRUPT < NMININT){
 2          if (A==0) A++; else A--;
 3          Entrop[K] = (Entrop[K] << 5) ^ (Entrop[K] >> 27) ^ HARDTICK() ^
 4                      (Entrop[(K + 1) & (SIZEENTROPY - 1)] >> 31);
 5          K = (K + 1) & (SIZEENTROPY - 1);
 6                      **repeated lines 2 to 5 XX times **
 7          }
```

Figure 1: Gathering uncertainty from the instruction cache and branch predictor

The *Entrop* array gathers (part of) the uncertainty injected in the instruction cache and the branch predictor by the operating system interrupts. The content of the *Entrop* array is recorded in a file and the *Entrop* array is reinitialized to zero.

5.2 Estimating the amount of collected entropy/uncertainty

Standard evaluation of entropy fails $f(k)$ being the frequency of appearance of event k from a source, the standard definition of entropy of a random source is:

$$E = -\sum_{k} f(k)\ log_2\ f(k) \qquad (1)$$

Unfortunately, formula (1) does not allow practical evaluation of the entropy of sources with large entropy (in practice larger than 30 bits).

The source of external entropy of the HAVEG algorithm is the modification of the internal hardware volatile states by external events such as operating system interruptions. Thousands of hardware volatile states are touched and modified on each OS interruption.

For instance, on a Solaris UltraSparc II station, setting NMININT to 1 and SIZEENTROPY to 8 shows that the first million 8-word recorded by the HAVEG algorithm illustrated in Figure 1 were **all distinct**. This indicates that the entropy introduced by an operating system interrupts in the hardware volatile states is larger than 20 bits per OS interrupts, but does not allow further estimation of the real entropy range.

Empirical estimation of entropy/uncertainty In order to empirically estimate the range of entropy/uncertainty introduced collected by a HAVEG algorithm on each OS interrupt, we used the following method.

Fixing the size of the *Entrop* array to 65536, we determine a threshold for NMININT above which the content of *Entrop* array is consistently considered as practically uniformly distributed by our battery of tests. On each experiment, a 16 Mbytes file was collected i.e, 64 successive runs. Then our battery of tests for uniform distribution was run.

Using this empirical estimation, we found that, on all the current target platforms of HAVEGE [18], the HAVEG algorithm illustrated in Figure 1 allows to gather at least 8K-64K unpredictable bits in average per operating system interrupt (from 8K on Itanium/Linux to 64K on Solaris/Ultrasparc II). That is, at least a few hundred thousands of unpredictable random bits in less than one CPU second, i.e three to four orders of magnitude more than the entropy gathered by previously available entropy gathering techniques.

6 HAVEGE: combining entropy/uncertainty gathering and pseudo-random number generation

The previously available entropy gathering methods cannot accommodate applications for which very high volume (i.e megabits or gigabits) of random numbers are needed. The usual way to accommodate such applications is to use a pseudo-random number generator that has been seeded with a (relatively) short unpredictable random key. This approach suffers two potential weaknesses. First, the key is relatively short and might be compromised by brute force. Second, the pseudo-random algorithm might suffer a weakness unknown to the user, but known to an attacker that might allow him/her to predict the sequence with a reasonable accuracy. The HAVEG algorithm family could be used to seed a pseudo-random number generator with longer unpredictable seeds or to periodically reseed a pseudo-random number generator.

As an alternative, we propose a more secure approach; the HAVEGE algorithm (HArdware Volatile Entropy Gathering and Expansion). HAVEGE combines concurrent continuous hardware volatile entropy gathering with pseudo-random number generation. This approach allows very high throughput of unpredictable random numbers (more than 100 Mbits/s on our target platforms) and offers a very high security level. At any step, the internal state of the HAVEGE random number generator consists of the content of large memory table and of thousands of internal hardware states. Security of the generator is first based on the inability for the user or an observer to directly monitor the internal state of the generator without altering it. Second, operating system interruptions and external events continuously reseed the generator.

6.1 Algorithm presentation

We modified the HAVEG algorithm to generate on-the-fly random numbers through a very simple algorithm. Two concurrent self-modifying walks in a table, while the collected entropy continuously updates the walk table.

The HAVEG algorithm family exercises unmonitorable hardware states in a predetermined order. Therefore long after the last interruption the content of the exercised unmonitorable hardware states is highly predictable (for instance the content of branch predictor entry). In the HAVEGE algorithm, the volatile hardware states are visited in chaotic order and/or are maintained in unpredictable states.

HAVEGE description An HAVEGE [8] algorithm is illustrated on Figure 2. It has been designed to fully activate both instruction and data L1 caches. It also activates a significant part of the branch prediction tables.

```
int Walk[128*1024];
register int pt,PT,PT2;
register int i;
 0           INITIALIZATION of pt,PT2 and Walk[0,..,16383] with   ! through HAVEG
 0           unpredictable random numbers                         ! for instance
 1     loop{
 2           if (pt & 0x4000) X++;                                ! exercising the branch predictor
 3           if (pt & 0x8000) X++;                                ! idem
 4           PT=pt & 0x1fff; pt= Walk[PT];                        ! exercising the L1 data cache
 5           PT2=Walk[(PT2 & 0xfff)^((PT ^ 0x1000) & 0x1000)];    ! idem
 6           RESULT[i]=PT2^pt;                                    ! hiding the walk content
 7           T = ((T << 7) + HardClock()) ^ (T >> 25);            ! reading the hardware clock counter
 8           pt = pt ^ T;
 9           Walk[PT]= pt;                                        ! updating the Walk table
10           i++;
11           ** lines 2 to 9 repeated 101 times **                ! for using the whole instruction cache
12           X+= Walk[pt & 0x1ffff];                              exercising the data TLB
13     }
```

Figure 2: An HAVEGE algorithm (version for UltraSparc II and Pentium III)

First an initialization phase consists in filling a memory table *Walk* twice as large as the L1 data cache with unpredictable random numbers. This is equivalent to seed the generator. This phase can be done using an HAVEG algorithm for instance.

The HAVEGE main loop is described below:

- HardClock() is a function that reads and returns the hardware clock counter value.
- Two concurrent walks are performed in parallel in a table of 8K 4-byte integers. The table is twice as large as the L1 data cache. That is, if the walks are random then, for each of the reads, the probability of a hit in the L1 cache is very close to $\frac{1}{2}$.

[8]The implementation we distribute is slightly different. But the underlying principles of its design are those presented here.

- Two distinct table entries are read through indirect pointers on each step (Lines 4 and 5). They are exclusive-ORED to generate a random number (Line 6). The purpose of the exclusive-OR of the two data read on the *Walk* table is to hide the content of the *Walk* table from any possible observer. If we had used directly the data read in the data as a random number then an observer might have been able to follow the walk for a while and might try to guess (part of) the content of the table.
- Two data dependent tests (line 2 and line 3) were introduced in each iteration of the walk to make its behavior dependent on branch prediction information. For both branches, the probability of the branch being taken is $\frac{1}{2}$ if the content of the table is unpredictable.
- The number of the unrolled steps (101) in the main loop of HAVEGE was tuned to get the inner loop body code just fitting in the instruction cache (for both Pentium III and UltraSparc II)[9]. This maximizes the number of instruction blocks (and associated branch prediction information) removed from the instruction cache on each operating system interruption.
- Line 12 in the algorithm was added to fully activate the data TLB. 128 pages are potentially touched by this instruction, i.e twice as the number of entries on the UltraSparc II and Pentium III. Probability of the accessed page of being present in the TLB is close to $\frac{1}{2}$. This last step in the algorithm is introduced to take part of the entropy introduced in the data TLB during operating system interruptions. However servicing a TLB miss is time consuming. To optimize the throughput of the generator, in a *while* iteration, the TLB is indirectly probed only once (the other accesses to the TLB will hit except just after the return from the interruption). This is sufficient to gather all the entropy injected in TLB by an OS interrupt.

6.2 Internal state of the HAVEGE generator

With a pseudo-random number generator, at any step in the computation, one can define its internal state as the values of internal variables and tables. This internal state determines the future behavior of the generator and the sequence of numbers that it will generate.

At any moment, one can also define the internal state of the HAVEGE generator as the content of the *Walk* table, the values of PT and $PT2$ pointers and the values of all volatile hardware states in the processor (branch predictors, instruction and data caches, ...), on the system bus and in the memory system that are touched by HAVEGE. This determines the execution time of the sequences of instructions in HAVEGE. If one was able to capture this internal state at a given point then he/she would be (theoretically) able to replicate the generated sequence from this point until the next occurrence of an interruption or until a new external event on the system bus or the memory system. However, collecting the internal state of the HAVEGE generator is infeasible at user-level, and any external event modifies it.

[9] This unrolling factor depends on the compiler and on the compiler options

We describe below part of the volatile information that belong to the internal state of the HAVEGE generator on an UltraSparc II workstation. A very similar analysis can be done for the Pentium III.

L1 Data cache From the HAVEGE generator, each of the 512 32-byte cache lines of the L1 data cache can be in one of seven possible states. Either it maps one of the two possible 32-byte blocks A0 or A1 from the *Walk* table or it maps a block irrelevant from HAVEGE. If it maps a relevant block then it maps only one or both the 16-byte sub-blocks. That is, the L1 cache is in one of 7^{512} possible states.

Data TLB The loop in HAVEGE touches 128 memory pages. From the HAVEGE algorithm perspective, an entry in the data TLB has 129 possible states: mapping one of the 128 pages or mapping another (irrelevant) page.

The data TLB on the Ultrasparc features 64 entries and is fully associative. From the HAVEGE generator perspective, the internal hardware state is represented by an ordered set of 64 pages. In normal mode, i.e long after the last interruption, all these pages belong to the *Walk* table and there are $\frac{128!}{64!}$ possible combinations.

L1 instruction cache The inner loop body in the self modifying walk (just) fits in the L1 instruction cache. The L1 instruction cache is two-way set-associative and branch prediction is embedded in the instruction cache. It features 256 sets.

Only two instruction blocks *I0* and *I1* from the main HAVEGE loop are mapped onto a given set s of the instruction cache. A LRU replacement policy is implemented on the instruction cache.

Therefore, from the HAVEGE perspective, each set has 7 different possible states, B being an instruction block independent from the HAVEGE algorithm: (*I0*, *I1*), (*I1*, *I0*), (B, *I0*), (B, *I1*), (*I0*, B), (*I1*, B) and (B, B). Then the instruction cache can be in at least 7^{256} different states.

On the UltraSparc II, the instruction cache also handles the branch prediction for both targets and directions. For targets,, two states are visible: either the target is correct or it is not. In the main loop in HAVEGE, there are 202 +1 conditional branches and 101 calls. From the HAVEGE perspective, the target prediction generator features at least 2^{304} different states. A 2-bit counter is associated with each conditional branch, therefore from the HAVEGE perspective, the branch predictor can be in 2^{406} different states.

Summary Considering only the three memorization structures above, it appears that the internal state of the HAVEGE generator includes thousands of binary internal volatile hardware states. From the analysis in Section 4, it appears that a significant fraction of these states are destructed (modified) by an interruption. Each of these states influences the self modifying walks in HAVEGE.

Moreover, many other volatile hardware states (pipeline states, buffer contents and status, L2 caches, etc.) are part of the HAVEGE internal state. Some of the volatile hardware states touched by an operating system interrupt that are part of the internal HAVEGE state are correlated. For instance when a instruction block is evicted then, its associated branch prediction information is also evicted. The next instruction block has also a high probability to be evicted. Nevertheless, there is little correlation between the blocks evicted from the data cache and the blocks evicted from the instruction cache, or between the branch prediction information destructed and the pages evicted from the data TLB.

6.3 Security of HAVEGE

The security of the HAVEGE generator relies on both the unfeasibility of reproducing its internal state, and on the continuous and unmonitorable injection of new uncertainty in its internal state by external events.

First, as pointed out in Section 3.2, there does not exist any user-level mean for the user itself to collect the precise internal volatile states of the processor at a given point. Therefore, nobody, even the user can access the global internal state of the HAVEGE generator. To reproduce the sequence of the generator after a given point (in the absence of new random states injection), one would have to reinitiate the algorithm with its complete internal state i.e, e contents of the table, internal variables and the pointers, but also the internal volatile hardware states. This task is intractable for the attacker. Collecting the hardware state of the processor requires freezing the hardware clock on the machine while running the random number generator:
If an attacker has got this right on your machine, then don't even think about protecting your data !.

Second, the knowledge of the internal state of the HAVEGE generator on a given cycle is not sufficient to reproduce the sequences. The internal state is also continuously touched by all the events on the memory system, on the bus system and by the operating system interrupts. Then even if the external observer had been able to capture the internal state of the generator at a point then he/she would only be able to follow the walk for a (very) limited delay unless he/she is also able to guess (monitor) all the new states continuously injected by external events. This has to be contrasted with the case of usual pseudo-random number generation. For a pseudo-random number generator, whenever the internal state is compromised at a point, the complete future sequence is compromised until new reseeding is explicitly performed .

6.4 Practical Uniform distribution

We checked that the sequences generated by the HAVEGE algorithm consistently pass our battery of tests for uniform distribution for all supported target platforms [18].

The tests were performed with the following protocol. The *Walk* table is initialized with unpredictable numbers through the HAVEG algorithm for gathering entropy from instruction cache and branch prediction tables presented in Section 5. We also checked that the content of the *Walk* table remains uniformly distributed, even after generating random numbers for a long time.

6.5 Performance

We checked the performance of the presented HAVEGE algorithm on several configurations for both Pentium III(II) Linux and UltraSparc II Solaris. In average on Pentium III, 920 million +/- 5 % cycles were needed to collect 32 Mbytes of random numbers, while on the UltraSparc II, 500 million +/- 5 % cycles were sufficient. This results in a throughput close to 280 Mbits/s on a 1Ghz Pentium III system and in a throughput of 150 Mbit/s on our (3-years old) 300 Mhz UltraSparc II. This throughput is 6 to 7 orders of magnitude larger than the throughput of the */dev/random* driver on Linux.

7 Conclusion

In this paper, we have presented, HAVEGE, a new heuristic to build very high performance software unpredictable random number generators.

Modern superscalar processors feature a lot of hardware components whose states do not influence the semantics of "normal" applications, but that have been added to enhance performance (caches, branch predictors, intermediate buffers, ...). They are also using complex operating systems. Interactions between user applications and the operating system create uncertainty in the processor states. This uncertainty is very significant within the memory hierarchy and the branch prediction structures. We have shown that, the hardware clock counter gives opportunity to indirectly gather this uncertainty and to produce unpredictable random number sequences: tens of thousands of unpredictable bits can be collected per operating system interruption.

HAVEGE packs pseudo-random number generation and entropy gathering on hardware volatile states within a simple code. The sequences generated by HAVEGE are unpredictable. Reproducing the sequence would necessitate to replicate the internal state of the generator, but this internal state consists in part of volatile hardware states. The generator is also continuously fed with new inputs on every interrupt. Reseeding is therefore automatic. Moreover no one, even the user itself, is able to access the complete internal state (seed) of the generator, since any attempt (except freezing the hardware clock) to access the volatile hardware states will alter these volatile states. To the best of our knowledge, the sequences are also unbiased since they pass an important battery of tests checking uniform distribution.

The throughput of the HAVEGE generator is very high (more than 100 Mbit/s). This means that producing unpredictable random number will consume less resources. Applications from other domains than cryptography may benefit from a high performance random number generator exhibiting very reliable uniform distributions.

Among the other advantages of HAVEGE, let us point out that the implementation is very simple and portable. Developing variations of HAVEGE for other processor architectures, other operating systems and other compilers is straightforward. One has only to adapt a few parameters related to instruction and data cache sizes and branch predictor sizes. The program is implemented at user level and does not rely on any operating system call. This is an important point in cryptography: you don't need to trust the developer if you can check the source code. Furthermore, while to the best of our knowledge the generated sequences do not suffer any exploitable bias, robustness of the HAVEGE generator could be further increased by combining it with other algorithmic pseudo-random number generation.

The technological trend in computer design is to use more and more complex processors featuring out-of-order execution and new hardware mechanisms for speculative execution, for memory (in)dependency prediction [13, 1, 16] as well as on-chip thread parallelism [20, 3, 4]. This will further create new uncertainty in the internal states of the processor and also create new opportunities to propagate this uncertainty. At the same time, new functionalities are also added in operating systems, therefore each operating system interruption will touch an always increasing set of volatile hardware states. Therefore, our approach for generating unpredictable random numbers on PCs and workstations will remain valid in the foreseeable future.

Status of HAVEGE (oct. 2002)

HAVEGE is distributed for tests and evaluation:
http://www.irisa.fr/caps/projects/hipsor/HAVEGE.html Currently supported platforms are:

- Solaris: UltraSparc I and II, UltraSparc III
- Linux: Pentium III, Pentium 4, Athlon and Itanium
- Windows: Pentium III, Pentium 4, Athlon
- MacOS10 and PowerPC G4

References

[1] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, pages 142–154, June 1998.

[2] D. Davis, R. Ihaka, and P. Fenstermacher. Cryptographic randomness from air turbulence in disk drives. *Lecture Notes in Computer Science*, 839:114–120, 1994.

[3] K. Diefendhorff. Compaq chooses SMT for Alpha. *Microprocessor Report*, Dec 1999.

[4] K. Diefendhorff. Power4 focuses on memory bandwidth. *Microprocessor Report*, Oct 1999.

[5] Intel Platform Security Division. The Intel random number generator. Technical report, Intel, 1999.

[6] A. Rukhin et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. *NIST special publication 800-22*, May 2001.

[7] R.W. Hamming. *Coding and Information Theory*. Prentice-Hall, 1980.

[8] IETF. RFC 1750 : Randomness recommendations for security, 1994.

[9] M. Jakobsson, E. Shriver, B. Hillyer, and A. Juels. A practical secure physical random bit generator. In *Proceedings of the 5th ACM Conference on Computer and Communications Security, November, 1998, San Francisco, pp. 103-111.*, 1998.

[10] T. Jennewein, U. Achleitner, G. Weihs, H. Weinfurter, and A. Zeilinger. A fast and compact quantum random number generator. *Preprint quant-ph/9912118*, 1999. `http://www.quantum.univie.ac.at/research/rng/`.

[11] Benjamin Jun and Paul Kocher. The intel random number generator. Cryptography Research, Inc. White Paper prepared for Intel Corporation, April 1999.

[12] J. Kelsey, B. Schneier, , and N. Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In H. Heys and C. Adams, editors, *Selected Areas in Cryptography, SAC'99*, number 1758 in LNCS. Springer, 2000.

[13] Richard E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[14] D.E. Knuth. *The Art of Computer Programming, Volume 2 / Seminumerical Algorithms.* Addison-Wesley, 1969.

[15] George Marsiglia. Diehard. `http://stat.fsu.edu/ geo/`.

[16] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 235–247, December 1997.

[17] Intel Platform Security Division Scott Durrant. Random numbers in data security systems. Technical report, Intel, 1999.

[18] André Seznec and Nicolas Sendrier. Havege: a user-level software unpredictable random number generator. Technical report, IRISA, 2002.

[19] A. Stefanov, N. Gisin, O. Guinnard, L. Guinnard, and H. Zbinden. Optical quantum random number generator. *Preprint quant-ph/9912118*, 1999. `http://www.gapoptic.unige.ch/Prototypes/QRNG/`.

[20] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, June 1995.

[21] John Walker. Hotbits: Genuine random numbers, generated by radioactive decay. `http://www.fourmilab.ch/hotbits/`.