HAVEGE: a User-level Software Heuristic for Generating Empirically Strong Random Numbers

André Seznec
IRISA-INRIA
and
Nicolas Sendrier
INRIA

Random numbers with high cryptographic quality are needed to enhance the security of cryptography applications. Software heuristics for generating empirically strong random number sequences rely on entropy gathering by measuring unpredictable external events. These generators only deliver a few bits per event. This limits them to being used as seeds for pseudo-random generators.

General-purpose processors feature a large number of hardware mechanisms that aim to improve performance: caches, branch predictors, ... The state of these components is not architectural (i.e. the result of an ordinary application does not depend on it). It is also volatile and cannot be directly monitored by the user. On the other hand, every operating system interrupt modifies thousands of these binary volatile states.

In this paper, we present and analyze HAVEGE (HArdware Volatile Entropy Gathering and Expansion), a new user-level software heuristic to generate practically strong random numbers on general-purpose computers. The hardware clock cycle counter of the processor can be used to gather part of the entropy/uncertainty introduced by operating system interrupts in the internal states of the processor. 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, it seems impossible even for the user itself to reproduce the generated sequences.

Categories and Subject Descriptors: G.3: probability and statistics [**C.1: processor architecture**]: D.4:operating system

General Terms: Cryptography, random number generation

Additional Key Words and Phrases: Superscalar processor, hardware clock counters

1. INTRODUCTION

The availability of a random number generator with high cryptographic qualities is one of the central issues of cryptographic implementations. This paper proposes HAVEGE (HArdware Volatile Entropy Gathering and Expansion), a new software

This work was supported by INRIA under the *Action de recherches coordonnées HIPSOR*
Authors addresses: André Seznec, IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France and Nicolas Sendrier, INRIA, Domaine de Voluceau, Rocquencourt, 78153 Le Chesnay Cedex, France

heuristic for generating "empirically strong random numbers" on general-purpose computers. We say that a random bit generator is empirically strong if there seems to be no practical way of distinguishing its output sequence from a sequence of independent random numbers uniformly distributed over $\{0, 1\}$. This is equivalent to saying that there seems to be no practical algorithm which, given part of the output sequence, can guess the next bit with success probability larger than $1/2$ (see, e.g., L'Ecuyer and Proulx [1989]). Of course, things would be more satisfactory if we could prove that no such algorithm exists. However, this type of proof exists for none of the practical random number generator proposed thus far. Note that the definition of an empirically strong random number generator implies that there seems to be no practical way to set this particular generator or any other generator in such a status that it will reproduce the same sequences.

The first contribution of this paper is the introduction of the HAVEG (HArdware Volatile Entropy Gathering) family of algorithms. General-purpose computers are built around processors featuring 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). Any external event involving an operating system interrupt or simply using the system bus and/or the memory system can induce the modification of hundreds or thousands of these non-architectural internal states. The HAVEG algorithms uses the hardware clock cycle counter as an indirect probe to extract (part of) the entropy/uncertainty introduced in the internal volatile hardware states by these external events. On current PCs and workstations, the HAVEG algorithms collects several tens of thousands of empirically strong random bits, on average, per every operating system interrupt, i.e., HAVEG is three to four orders of magnitude more efficient than previous software entropy gathering techniques.

The second contribution of the paper is the introduction of HAVEGE. HAVEGE combines HAVEG-like entropy gathering with simple pseudo-random number generation. The HAVEGE generator exhibits a very high throughput (in the same range as the UNIX *rand* pseudo-random function). The internal state of HAVEGE consists of classical data mapped in memory and in thousands of binary volatile hardware states (addresses of data blocks present in the cache, branch prediction table contents, ...). Most of these hardware states are maintained unpredictable by HAVEGE. 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. Therefore, reproducing the sequences produced by HAVEGE appears virtually impossible.

The remainder of the paper is organized as follows. In Section 2, we describe some of the states of internal components that influence 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 the external events. Section 3 introduces and analyzes the HAVEG family of algorithms. These algorithms gather the uncertainty introduced in internal volatile hardware states by operating system interrupts. Section 4 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 random numbers at a very high rate and continuously gather and propagate the uncertainty introduced by external events in internal hardware states. The internal invisible state of the HAVEGE generator is analyzed. Section 5 presents some empirical evidence of randomness of the sequences generated by HAVEG and HAVEGE. Section 6 briefly compares HAVEG and HAVEGE with the current practice of software generation of empirically strong random numbers. Finally, Section 7 summarizes this study.

2. MICROARCHITECTURE STATES AND THE EXECUTION TIME OF A SEQUENCE OF INSTRUCTIONS

In a computer system 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 external events to the process.

For instance the precise number of cycles for executing a simple sequence of instructions including at least one conditional branch and one load/store depends on branch prediction outcome and on the presence or absence of data and instructions in the different caches of the memory hierarchy. 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 and in numerous buffers in the processor.

The status of all these internal states depends on the past execution on the processor. These states are affected by events external to the process such as operating system interrupts or transactions on the system bus. Seznec and Sendrier [2002] showed on current workstations and PCs thousands of hardware volatile states are modified in the data/instruction caches and branch predictors on each operating system interrupt.

General purpose instruction sets do not provide user-level direct ways to monitor the internal states of the processor. 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! Therefore, the knowledge of the global state of a microprocessor for an arbitrary point in any program is unattainable for the owner of the process and *a fortiori* for the attacker.

However, the instruction sets of most modern microprocessor systems provide the user with a direct access to the hardware clock counter. Reading the hardware clock counter can be used as an indirect probe to collect information on the internal hardware states in the processor.

3. 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 interrupt. Therefore one can expect that significant uncertainty is injected in these volatile hardware states on each operating system interrupt.

We present a simple family of entropy gathering algorithms, the HAVEG (HArdware Volatile Entropy Gathering) algorithms. Unlike previous entropy gathering implementations, HAVEG algorithms are run 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 activate different components in a microprocessor.

An HAVEG algorithm designed for gathering uncertainty from the instruction cache and the branch prediction structure is illustrated in Figure 1 and detailed below.

`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 by function `HARDTICK()` whenever this difference is higher than a threshold indicating that there was an interrupt between two successive reads of the hardware clock counter.

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. There is much more entropy/uncertainty in the least significant bits of the hardware clock counter than in the most significant ones. Therefore on line 3, we combine this read value with a circular shift of the previously accumulated data. Moreover, the uncertainty is also diffused on the whole array `Entrop` through combination through exclusive-or and shift with the next element in array (line 4).

The unrolling factor (XX) should be adjusted for each of the targeted architectures: XX is chosen in order to maximize the size of the compiled loop body without exceeding the size of the instruction cache and without overflowing the branch prediction structures sizes.

The while loop is run until the number of encountered interrupts 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 interrupt present much more uncertainty than the iterations that occur long after the last interrupt.

```
int Entrop[SIZEENTROPY];
int A;
  1   while (NBINTERRUPT < 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                       **repeat lines 2 to 5 XX times **
  7          }
```

Fig. 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 saved and the `Entrop` array is reinitialized to zero.

4. HAVEGE: COMBINING ENTROPY/UNCERTAINTY GATHERING AND PSEUDO-RANDOM NUMBER GENERATION

In order to accommodate applications requiring very high volume of random numbers, we modified the HAVEG algorithm to generate on-the-fly random numbers through a very simple algorithm.

HAVEGE (HArdware Volatile Entropy Gathering and Expansion) combines concurrent continuous hardware volatile entropy gathering with pseudo-random number generation. The presented implementation uses a very simple pseudo-random number generator: two concurrent self-modifying walks in a single table are run while the collected entropy continuously updates the walk table.

4.1 Algorithm presentation

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

An HAVEGE algorithm is illustrated on Figure 2. It has been designed to fully activate both instruction and data L1 caches. This algorithm can be adapted to any processor featuring an instruction cache and a data cache. The illustrated algorithm also activates a significant part of the branch prediction tables.

```
int Walk[2*CACHESIZE];
register int pt, PT, PT2;
register int i;
 0          INITIALIZATION of pt, PT2 and Walk[0,..,2*CACHESIZE-1] with
 0          empirically strong random numbers through HAVEG
 1    loop{
 2          if (pt & 8*CACHESIZE) X++;
 3          if (pt & 16*CACHESIZE) X++;
 4          PT=pt & (2*CACHESIZE-1); pt= Walk[PT];
 5          PT2=Walk[(PT2 & CACHESIZE-1)^((PT ^ CACHESIZE) & CACHESIZE)];
 6          RESULT[i]=PT2 ^pt;
 7          T = ((T << 7) + HardClock()) ^ (T >> 25);
 8          pt = pt ^ T;
 9          Walk[PT]= pt;
10          i++;
11          ** repeat lines 2 to 9 YY times **
12    }
```

Fig. 2. An HAVEGE algorithm

First an initialization phase consists in filling a memory table `Walk` twice as large as the size `CACHESIZE` of the L1 data cache with empirically strong random numbers[1]. This is equivalent to seeding 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 4-byte integers. The table is twice as large as the L1 data cache. That is, if the walks are random then the probability of a hit in the cache is very close to 1/2 on each data reading in the table.

—Two distinct table entries are read through indirect pointers on each step (Lines 4 and 5). We take their bitwise exclusive-or 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 `Walk` table 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 1/2 if the content of the table is random.

—The number of unrolled steps (YY) in the main loop of HAVEGE should be tuned to get the inner loop body code just fitting in the instruction cache[2]. This maximizes the number of instruction blocks (and associated branch prediction information) removed from the instruction cache on each operating system interrupt.

4.2 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. An analysis presented by Seznec and Sendrier [2002] shows that this represents thousands of binary states on modern superscalar processors.

4.3 Reproducing HAVEGE sequences

In practice, 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.

[1]This implementation assumes that `CACHESIZE` is a power of two.
[2]This unrolling factor depends on the compiler and on the compiler options

First, as pointed out in Section 2, there does not exist any mean for the user itself to collect the precise internal volatile states of the processor at a given point. Therefore, nobody, not 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, the contents of the table, internal variables and the pointers, but also the internal volatile hardware states. In practice, this task is intractable for the attacker. Collecting the global 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.

5. EMPIRICAL EVIDENCE OF RANDOMNESS

In this section, we provide some empirical evidence on the performance of the HAVEG and HAVEGE generators in terms of statistical distribution of the generated sequences and in terms of throughput.

Proving that a sequence of bits is random is virtually impossible. However most non-random sequences fail some algorithmic tests. We first describe the battery of statistical tests that were used as empirical evidence of randomness for the generated sequences. Then we present an evaluation of the entropy/uncertainty collected by HAVEG. Finally, we present results of the statistical tests on HAVEGE and also the HAVEGE throughput on current hardware platforms.

5.1 Statistical tests for randomness

The battery of tests for randomness used as empirical evidence of randomness in this paper includes Lempel-Ziv compression test, entropy test, chi2 test, Monte Carlo tests from *ent*[3], the FIPS-140-2 test suite for random number generators [FIPS-140-2 2001] [4], the DIEHARD suite and the NIST statistical suite for random number generator [Rukhin et al. 2001].

The methodology we use is as follows. Sequences of 16 Mbytes of random numbers are collected and tested. Each of the four steps is run on several sequences (typically 10).

As a first step, we use the *ent* suite as a filter, since it is not CPU time consuming: we never encountered a 16-Mbyte sequence that was considered random by the other tests in the suite, but fails badly on *ent*. So whenever one 16-Mbyte sequence out of 10 fails badly on one of the *ent* tests, we reject the generator.

[3] available from `http://www.fourmilab.ch/random/`
[4] C code available from `http://people.qualcomm.com/ggr/QC/index.html`

The second step consists in applying the FIPS-140-2 test suite (runs tests and poker tests) on the sequences. In a 16-Mbyte sequence, 6400 non-overlapping 20000 bits sequences are tested. Testing pseudo-random number generators such as Mersenne twister [Matsumoto and Nishimura 1998] with recognized uniform distribution properties, we remarked that the cumulated number of failed tests over the 6400 non-overlapping 20000 bits sequences generally varies between 1 and 11 with exceptional cases with no failures and with 12 and 13 failures. Whenever discrepancies from these results are detected, (i.e, more rejected 20000 bits sequences in one or more 16-Mbyte sequences out of 10), we reject the generator.

In a third step, we apply the NIST statistical suite for random number generator. This suite consists in the 16 different statistical tests. Since the performance of the distributed suite is not adapted to test large random number sequences, we reengineer the suite while respecting all its functionalities. Tests were rewritten in order to reduce CPU time. Every test is applied, but the most time consuming tests are not applied on all non-overlapping subsequences, but only on a sample of these non-overlapping subsequences. Tests are run using 0.001 as a threshold. The NIST suite is run on 16-Mbyte random number files. For each 16-Mbyte file, it produces from 350 to 508 p-values. The generator is rejected whenever the number of failures (number of p-values less than 0.001) is constantly non-zero on a suite of 10 16-Mbyte files collected in a row.

In order to determine this criterion, we applied the test suite on a the Mersenne twister which is known to exhibit very reliable uniform distribution properties. In tests on 1000 successive 16-Mbyte sequences generated with different random seeds, we obtained a maximum of 5 non-null successive number of failures. and we encountered 730 sequences with no failure, 199 sequences with a single failure, 50 sequences with 2 failures, 12 sequences with 3 failures, 3 sequences with 4 failures, 4 sequences with 5 failures and 2 sequences with 6 failures.

However, in practice, we did not encounter HAVEG or HAVEGE-like generators that were borderline for the NIST statistical suite, i.e. either the generator was clearly passing the suite with only a few of the 10 16-Mbyte sequences encountering a few failures or every 16-Mbyte sequence was encountering a large number of failures (more than 10).

The DIEHARD suite is used as a final test. In practice in our experiments, we did not encounter any generator (related to HAVEGE and HAVEG) that passed the other tests and was rejected by DIEHARD[5].

Note that passing our battery of tests for randomness is not a proof of randomness. However, it may be considered as an indicator that finding and exploiting a bias in the generated sequences would be very difficult, particularly for a non-deterministic random number generator.

5.2 Estimating the amount of collected entropy/uncertainty through HAVEG

HAVEG is collecting entropy after each OS interrupt. The OS interrupt is the source of entropy. The `Entrop` array constitutes a single measure on this random

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

source and we would like to evaluate the entropy of this measure.

5.2.1 *Standard evaluation of entropy fails.* Let $f(k)$ be the probability of appearance of event k from a source, the standard definition of entropy of a random source is:

$$H = -\sum_{k} f(k)\ \log_2\ f(k) \tag{1}$$

Unfortunately, this formula does not allow to evaluate the entropy of a source with large entropy (in practice larger than 30 bits) because the required sample size would be too large.

5.2.2 *Empirical estimation of "entropy"/uncertainty.* In order to empirically estimate the range of "entropy" or uncertainty introduced in average by each OS interrupt that is collected by a HAVEG algorithm, 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 random by our statistical test suite. On each experiment, a 16-Mbyte file was collected corresponding to 64 successive runs.

Using this empirical estimation, we found that, on all the target platforms of HAVEGE in 2002 [Seznec and Sendrier 2002], the HAVEG algorithm illustrated in Figure 1 allows to gather at least 8K-64K random bits in average per operating system interrupt (from 8K on Itanium/Linux to 64K on Solaris/Ultrasparc II). That is, at least three to four orders of magnitude more than the "entropy" gathered by previously available entropy-gathering techniques.

Note that this empirical estimation provides an upper bound on the actual entropy that is collected by HAVEG, since the battery of tests may fail to detect some correlation or structures in the sequences. Then it might be considered safer to collect only a smaller number of bits per interruption than given by our empirical estimation.

5.3 Empirical evidence of randomness for sequences generated by HAVEGE

We checked that the sequences generated by the HAVEGE algorithm consistently pass our battery of statistical tests for randomness. This holds for all the supported target platforms [Seznec and Sendrier 2002].

The tests were performed with the following protocol. The *Walk* table is initialized with random numbers through the HAVEG algorithm presented in Section 3. We also checked that the content of the *Walk* table remains random, even after generating random numbers for a long time.

As an example, we illustrate the results of an extensive execution (100 runs) of the third step (the NIST statistical suite) of our test on the HAVEGE algorithm running on a Pentium 4 system under Linux operating system. Three days of CPU time on a Pentium 4 system were needed to complete this experiment. The NIST statistical suite was applied on 1000 16-Mbyte sequences generated by HAVEGE. A maximum of 5 consecutive 16-Mbyte sequences with one or more failures was encountered in this experiment. In the total experiment, i.e. on 1000 consecutive 16-Mbyte sequences generated by the described HAVEGE algorithm on a Pentium 4, we encountered 732 sequences with no failure, 195 sequences with a single failure, 45 sequences with 2 failures, 14 sequences with 3 failures, 7 sequences with 4 failures, 5 sequences with 5 failures and 2 sequences with 6 failures. The distribution of failures is very similar to the distribution of failures that we obtained on sequences generated with the Mersenne twister generator.

5.4 Throughput of HAVEGE

We checked the performance of the presented HAVEGE algorithm on several hardware configurations available in 2002 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 throughput is in the same range as the throughput of standard pseudo-random number generators.

6. RELATED WORK ON SOFTWARE GENERATION OF EMPIRICALLY STRONG RANDOM NUMBERS

Computers are built to execute usual software algorithms in a deterministic fashion. Some computer systems feature a dedicated hardware random number generator (e.g. [Jun and Kocher 1999]).

In the absence of such a hardware random number generator on a computer, the current practice is to measure parameters of some external events through software. This technique is known as entropy gathering. Many events (mouse, keyboard, disk, network, ...) occurring in a computer, though deterministic by nature, are sources of uncertainty. For instance, there is a possibility to exploit the randomness from chaotic air turbulence in disk drives through software timing of the accesses to particular sectors on the disk [Davis et al. 1994]. Jakobsson et al. [1998] claimed that only 5 random bits per minute 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 such as DIEHARD suite were unable to distinguish the output of the utility mode from truly random sequence.

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

Due to the limited throughput of former entropy gathering techniques, practical solutions in cryptography use a pseudo-random number generator periodically reseeded with random numbers obtained by entropy gathering techniques as for instance in Yarrow [Kelsey et al. 2000]. In these approaches, whenever the internal state of the pseudo-random number generator is compromised at a point, the complete future sequence is compromised until new reseeding is performed. This has to be contrasted with HAVEGE which is continuously reseeded by all the external events.

Among the other advantages of HAVEG and HAVEGE over previous entropy gathering techniques, let us point out that the implementation is very simple and portable. Porting one of the previous entropy gathering techniques to a new platform (architecture and/or operating system) is a major effort since it generally involves developments in the operating system. By contrast, 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.

7. CONCLUSION

In this paper, we have presented HAVEGE, a new heuristic to build high performance, software-based and empirically strong random number generators for computer systems built around modern superscalar processors.

HAVEGE packs pseudo-random number generation and entropy gathering on hardware volatile states in a single code. Reproducing the sequence would require 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, not 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. The sequences pass an important battery of tests checking randomness.

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 pseudo-random number generators.

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)dependence prediction [Kessler 1999; Chrysos and Emer 1998; Moshovos and Sohi 1997] as well as on-chip thread parallelism [Tullsen et al. 1996; Diefendhorff 1999a; 1999b]. 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 interrupt will touch an increasing set of volatile hardware states. Therefore, our approach for generating empirically strong random numbers on PCs and workstations will remain valid in the foreseeable future.

Acknowledgements

The authors would like to thank Matthieu Lemoine and Assia Djabelkhir for their contributions in the early phase of this study. They would also like to thank Pierre L'Ecuyer, editor of this special issue, for his help in polishing the final version of the paper.

REFERENCES

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

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

Diefendhorff, K. 1999a. Compaq chooses SMT for Alpha. *Microprocessor Report*.

Diefendhorff, K. 1999b. Power4 focuses on memory bandwidth. *Microprocessor Report*.

FIPS-140-2. 2001. Security requirements for cryptographic modules. *Federal Information Processing Standard publication 140-2*.

JAKOBSSON, M., SHRIVER, E., HILLYER, B., AND JUELS, A. 1998. 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.*

JUN, B. AND KOCHER, P. 1999. The intel random number generator. Cryptography Research, Inc., White Paper prepared for Intel Corporation.

KELSEY, J., SCHNEIER, B., AND FERGUSON, N. 2000. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Selected Areas in Cryptography, SAC'99*, H. Heys and C. Adams, Eds. Number 1758 in LNCS. Springer.

KESSLER, R. E. 1999. The Alpha 21264 microprocessor. *IEEE Micro 19,* 2, 24–36.

L'ECUYER, P. AND PROULX, R. 1989. About polynomial-time unpredictable generators. In *Proceedings of the 1989 Winter Simulation Conference*. IEEE Press, 467–476.

MATSUMOTO AND NISHIMURA. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation 8,* 1, 3–30.

MOSHOVOS, A. AND SOHI, G. S. 1997. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*. 235–247.

RUKHIN, A., SOTO, J., NECHVATAL, J., SMID, M., BARKER, E., LEIGH, S., LEVENSON, M., VANGEL, M., BANKS, D., HECKERT, A., DRAY, J., AND VO, S. 2001. A statistical test suite for random and pseudorandom number generators for cryptographic applications. *National Institute of Standards and Technology publication 800-22*.

SEZNEC, A. AND SENDRIER, N. 2002. Hardware volatile entropy gathering and expansion: generating unpredictable random number at user level. Tech. Rep. 4592, INRIA.

TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. 1996. Exploiting choice : Instruction fetch and issue on an implementable simultaneous MultiThreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*. ACM Press, New York, 191–202.