HAVEGE

HArdware Volatile Entropy Gathering and Expansion

Unpredictable random number generation at user level

André Seznec

Nicolas Sendrier

Unpredictable random numbers

- Unpredictable = irreproducible + uniformly distributed

- Needs for cryptographic purpose:
  - ➔ key generation, paddings, zero-knowledge protocols, ..

- Previous solutions:
  - ➔ **hardware: exploiting some non deterministic physical process**
    - **10-100 Kbits/s**
  - ➔ **software: exploiting the occurences of *(pseudo)* non deterministic external events**
    - **10-100 bits/s**

André Seznec
Caps Team
Irisa

Previous software entropy gathering techniques

- Gather entropy from a few parameters on the occurences of various external events:
  - mouse, keyboard, disk, network, ..

- But ignore the impacts of these external events in the processor states

HAVEGE:
HArdware Volatile Entropy Gathering and Expansion

Thousands of hardware states for performance improvement in modern processors

These states are touched by all external events

Might be a good source of entropy/uncertainty !

André Seznec
Caps Team
Irisa

HAVEGE:

HArdware Volatile Entropy Gathering ***and Expansion***

HAVEGE combines in the same algorithm:

- gathering uncertainty from hardware volatile states

. a few 100Kbits/s

- pseudo-random number generation

. more than 100 Mbits/s

André Seznec
Caps Team
Irisa

Hardware Volatile States in a processor

- States of many microarchitectural components:
  - caches: instructions, data, L1 and L2, TLBs
  - branch predictors: targets and directions
  - buffers: write buffers, victim buffers, prefetch buffers, ..
  - pipeline status

A common point

these states are volatile and not architectural:

-the result of an application does not depend of these states

-these states are unmonitorable from a user-level application

An example: the Alpha 21464 branch predictor

- 352 Kbits of memory cells:
  - ➔ indexed by a function of the instruction address + the outcomes of more than 21 last branches
- on any context switch:
  - ➔ inherits of the overall content of the branch predictor

Any executed branch lets a footprint on the branch predictor

André Seznec
Caps Team
Irisa

Gathering hardware volatile entropy/uncertainty ?

Collecting the complete hardware state of a processor:

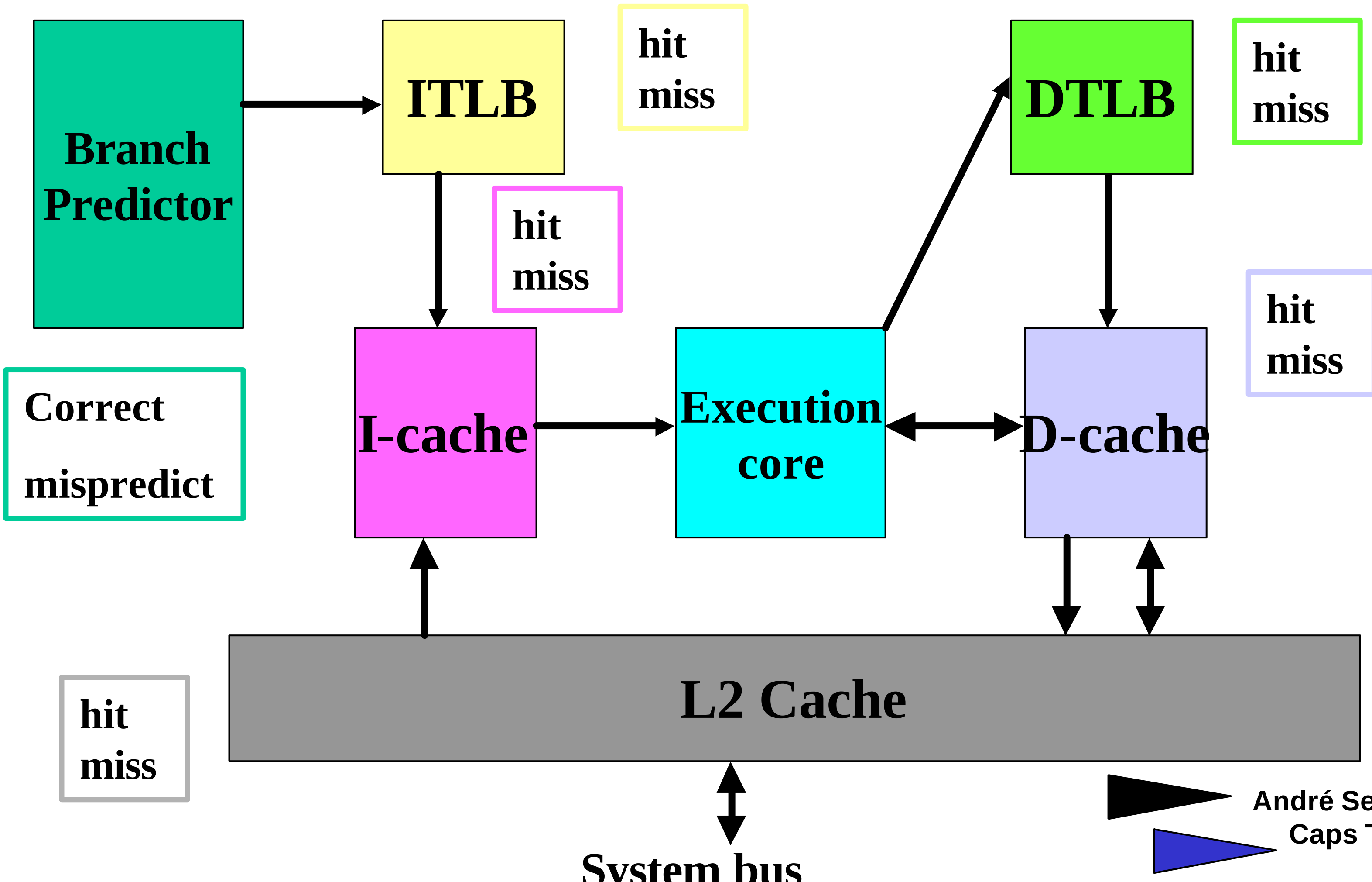
- **requires freezing the clock**
- not accessible on off-the-shelf PCs or workstations

Indirect access through timing:

- use of the hardware clock counter *at a very low granularity*
- Heisenberg 's criteria:

  indirect access to a particular state (e.g. status of a branch predictor entry) modifies many others

Execution time of a short instruction sequence is a complex function !



André Seznec
Caps Team
Irisa

Execution time of a short instruction sequence is a complex function (2) !

- state of the execution pipelines:
  - ➔ up to 80 instructions inflight on Alpha 21264, more than 100 on Pentium 4

- precise state of every buffer
- occurrence on any access on the system bus

But a processor is built to be deterministic !?!

Yes but:

- Not the response time !
- External events: peripherals , IOs
- Operating System
- Fault tolerance

OS interruptions and some volatile hardware states
Solaris on an UltraSparc II (non loaded machine)

- L1 data cache: 80-200 blocks displaced
- L1 instruction cache: around 250 blocks displaced
- L2 cache: 850-950 blocks displaced
- data TLB: 16-52 entries displaced
- instruction TLB: 6 entries displaced

Thousands of modified hardware states

- + that 's a minimum
- + distribution is erratic

Similar for other OS and other processors

HArdware Volatile Entropy Gathering

example of the I-cache + branch predictor

```
While (INTERRUPT < NMININT){
        if (A==0) A++; else A--;

        Entrop[K]= (Entrop[K]<<5) ^ HardTick () ^ (Entrop[K]>>27) ^
                  (Entrop[(K+1) & (SIZEENTROPY-1)] >>31;

        K= (K+1) & (SIZEENTROPY-1);
                  ** repeated XX times **
}
```

Gather through several OS interruptions

Exercise the branch prediction tables

Gathering uncertainty in array Entrop

Exercising the whole I-cache

HArdware Volatile Entropy Gathering

I-cache + branch predictor (2)

- The exact content of the Entrop array depends on the exact timing of each inner most iteration:
  - presence/absence of each instruction in the cache
  - status of branch prediction
  - status of data (L1, L2, TLB)
  - precise status of the pipeline
  - activity on the data bus
  - status of the buffers

Estimating the gathered uncertainty

- The source is the OS interruption:
  - ➔ width of the source is thousands of bits
  - ➔ no practical standard evaluation if entropy is larger than 20

<u>1M samples of 8 words after a single interrupt were all distinct</u>

- Empirical evaluation: NIST suite + Diehard
  - ➔ consistantly passing the tests = uniform random

Uncertainty gathered with HAVEG on unloaded machines

- Per OS interrupt in average and depending on OS + architecture
  - ➔ 8K-64K bits on the I-cache + branch predictor
  - ➔ 2K-8K bits on the D-cache
- A few hundred of unpredictable Kbits/s
  - ➔ 100-1000 times more than previous entropy gathering techniques on an unloaded machine

HAVEG algorithms and loaded machines

- On a loaded machine:
  - ➔ more frequent OS interrupts:
    - less iterations between two OS interrupts
  - ➔ less uncertainty per interrupt
    - i.e., more predictable states for data and inst. caches

- But more uncertainty gathered for the same number of iterations :-)

HAVEG algorithms and loaded machines (2)

Determine the number of iterations executed on a non-loaded machine

```
for (i=0;i<EQUIVWORKLOAD;i++){
        if (A==0) A++; else A--;
        Entrop[K]= (Entrop[K]<<5) ^ HardClock () ^ (Entrop[K]>>27) ^
                  (Entrop[(K+1) & (SIZEENTROPY-1)] >>31;
        K= (K+1) & (SIZEENTROPY-1);
                  ** repeated XX times **
}
```

André Seznec
Caps Team
Irisa

Reproducing HAVEG sequences ?

Security assumptions

- An attacker has user-level access to the system running HAVEG
  - ➔ He/she cannot read the memory of the HAVEG process
  - ➔ He/she cannot freeze the hardware clock
  - ➔ He/she cannot hardware monitor the memory/system bus
- An attacker has unlimited access to a similar system (hardware and software)

Heisenberg's criteria

Nobody, not even the user itself can access the internal volatile hardware state without modifying it

Passive attack: just observe, guess and reproduce (1)

- Need to « guess » (reproduce) the overall initial internal state of HAVEG:
  - ➔ the precise hardware counter ?
  - ➔ the exact content of the memory system, disks included !
  - ➔ the exact states of the pipelines, branch predictors, etc
  - ➔ the exact status of all operating system variables

Without any internal dedicated hardware on the targeted system ?

Passive attack: just guessing and reproducing (2)

- reproducing the exact sequence of external events on a cycle per cycle basis
  - ➔ network, mouse, variable I/O response times, …
  - ➔ internal errors ?

Without any internal dedicated hardware on the targeted system ?

Active attack: setting the processor in a predetermined state

- Load the processor with many copies of a process that:
  - flushes the caches (I, D, L2 caches)
  - flushes the TLBs
  - sets the branch predictor in a predetermined state
- HAVEG outputs were still unpredictable

HAVEG vs usual entropy gathering

- User level
- automatically uses every modification on the volatile states

- Embedded in the system
- measures a few parameters

There is more information in a set of elements than in the result of a function on the set

HAVEGE

HAVEG and Expansion

André Seznec
Caps Team
Irisa

HAVEG is CPU intensive

- The loop is executed a large number of times, but **long after** the last OS interrupt, hardware volatile states tend to be in a predictable state:
  - ➔ instructions become present in the cache
  - ➔ branch prediction information is determined by the N previous occurrences
  - ➔ presence/absence of data in the data cache is predictable

Less uncertainty is gathered long after the last OS interrupt

HAVEGE= HAVEG + pseudo-random number generation

Embed an HAVEG-like entropy gathering algorithm in a pseudo-random number generator

A very simple PRNG:

- **two concurrent walks in a table**
- **random number is the exclusive-OR of the two read data**

But the table is continuously modified using the hardware clock counter

An example of inner most iteration

```
if (pt & 0x4000){ PT2 = PT2 ^ 1;}
if (pt & 0x8000){ PT2 = PT2 + 7;}

PT=pt & 0x1fff; pt= Walk[PT];
PT2=Walk[(PT2 & 0xfff) ^
          ((PT ^ 0x1000) & 0x1000)];

RESULT[i] ^ =  PT2 ^ pt  ; i++;

T=((T<< 11) ^ (T>> 21)) + HardClock();
pt = pt ^ T; Walk[PT]= pt;
```

Tests to exercise the branch predictor

The two concurrent walks

Output generation

Entropy gathering and table update

André Seznec
Caps Team
Irisa

HAVEGE loop

- Number of unrolled iterations is adjusted to fit exactly in the instruction cache:
  - ➔ exercise the whole I-cache and the branch prediction structure
- Size of the table is adjusted to twice the data cache size:
  - ➔ hit/miss probability is maintained close to 1/2
- + a few other tricks:
  - ➔ exercise the TLB
  - ➔ personalize each iteration

HAVEGE internal state

The usual memory state of any PRNG

+

Internal volatile hardware states:	On a Solaris UltraSparcII
branch predictor	(2**406) * (2**304) states
I-cache	7**256 states
data cache	7**512 states
data TLB	128!/64! States
miscelleanous, ..	..

André Seznec
Caps Team
Irisa

Maintaining unpredictable *hidden* volatile states

```
if (pt & 0x4000){ PT2 = PT2 ^ 1;}
if (pt & 0x8000){ PT2 = PT2 + 7;}

PT=pt & 0x1fff; pt= Walk[PT];
PT2=Walk[(PT2 & 0xfff) ^
          ((PT ^ 0x1000) & 0x1000)];

RESULT[i] ^ =  PT2 ^ pt  ; i++;

T=((T<< 11) ^ (T>> 21)) + HardClock();
pt = pt ^ T; Walk[PT]= pt;
```

Taken or not-taken with p = 1/2

Hit/miss on the L1 cache with p = 1/2

Security of HAVEGE= internal state

- Reproducing HAVEGE sequences:
  - ➔ internal state is needed
- Collecting the internal state is impossible:
  - ➔ destructive
  - ➔ *or freezing the hardware clock !*

- If an attacker was able to capture (guess) a valid internal state then he/she must also monitor (guess) all the new states continuously injected by external events

Dealing with continuous and unmonitorable reseeding is not easy !!

HAVEGE continuous reseeding

- On each OS interrupt:
  - ➔ internal state of the generator is modified
    - thousands of binary states are touched
  - ➔ complex interaction between internal general state and OS servicing:
    - service time of an OS interrupt depends on the initial hardware state
- Any event on the memory system touches the state
  - ➔ asynchronous events on the memory bus !

André Seznec
Caps Team
Irisa

HAVEGE:
uniform distribution and irreproducibility

- When the *Walk* table is initialized with uniformly distributed random numbers, generated numbers are uniformly distributed
  - ➔ use of an initialization phase: HAVEG

- Irreproducibility:
  - ➔ irreproducibility of the initial state ensures irreproducibility of the sequences
  - ➔ even, with the same initial *Walk* table content, rapid divergence of the result sequences:
    - collecting the ith to i+16th results pass the tests for i= 100000

HAVEGE 1.0

- Initialization phase 1:
  - ➔ HAVEG on instruction cache and branch predictor
- Initialization phase 2:
  - ➔ HAVEGE without result production

One CPU second worth recommended per phase

To our knowledge 1/20s and a single phase is sufficient

- HAVEGE main loop

Portability

- User level
  - ➔ access to the hardware clock counter in user mode is needed
- Just adapt a few parameters:
  - ➔ I and D cache size, branch predictor sizes
  - ➔ adjust the number of iterations in the loops to fit the I-cache

Performances HAVEGE1.0

- To collect 32 Mbytes on unloaded machines:
  - ➔ 570 million cycles on UltraSparc II
  - ➔ 890 million cycles on Pentium III (gcc Linux and Windows)
  - ➔ 780 million cycles on Pentium III (Visual C++)
  - ➔ 1140 million cycles on Athlon (gcc Linux and Windows)
  - ➔ 1300 million cycles on Itanium

over 100 Mbits/s on all platforms

HAVEGE2.0

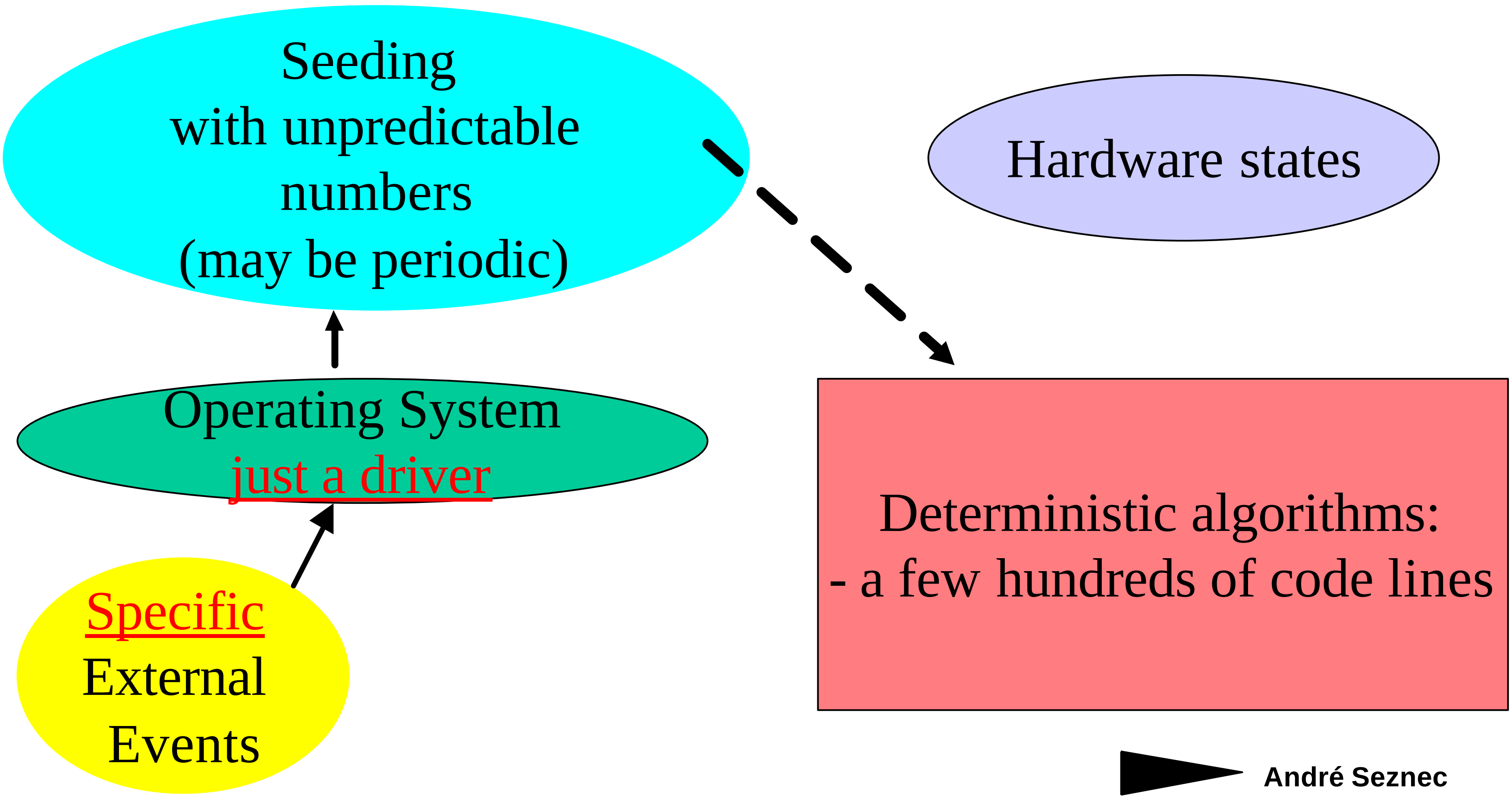
- Reengineered for :
  - ➔ Simplicity:
    - A single loop for initialization and production
  - ➔ Portability:
    - Setting the data cache, TB sizes
    - Adapting the number of iterations
  - ➔ Performance for non-cryptographic applications

Performances HAVEGE2.0 (non cryptographic)

- To collect 32 Mbytes on unloaded machines:
  - 260 million cycles on UltraSparc II
  - 270 million cycles on Pentium 4 (gcc Linux and Windows)
  - 270 million cycles on PowerPC 7400 (MacOS 10)
  - 630 million cycles on Itanium

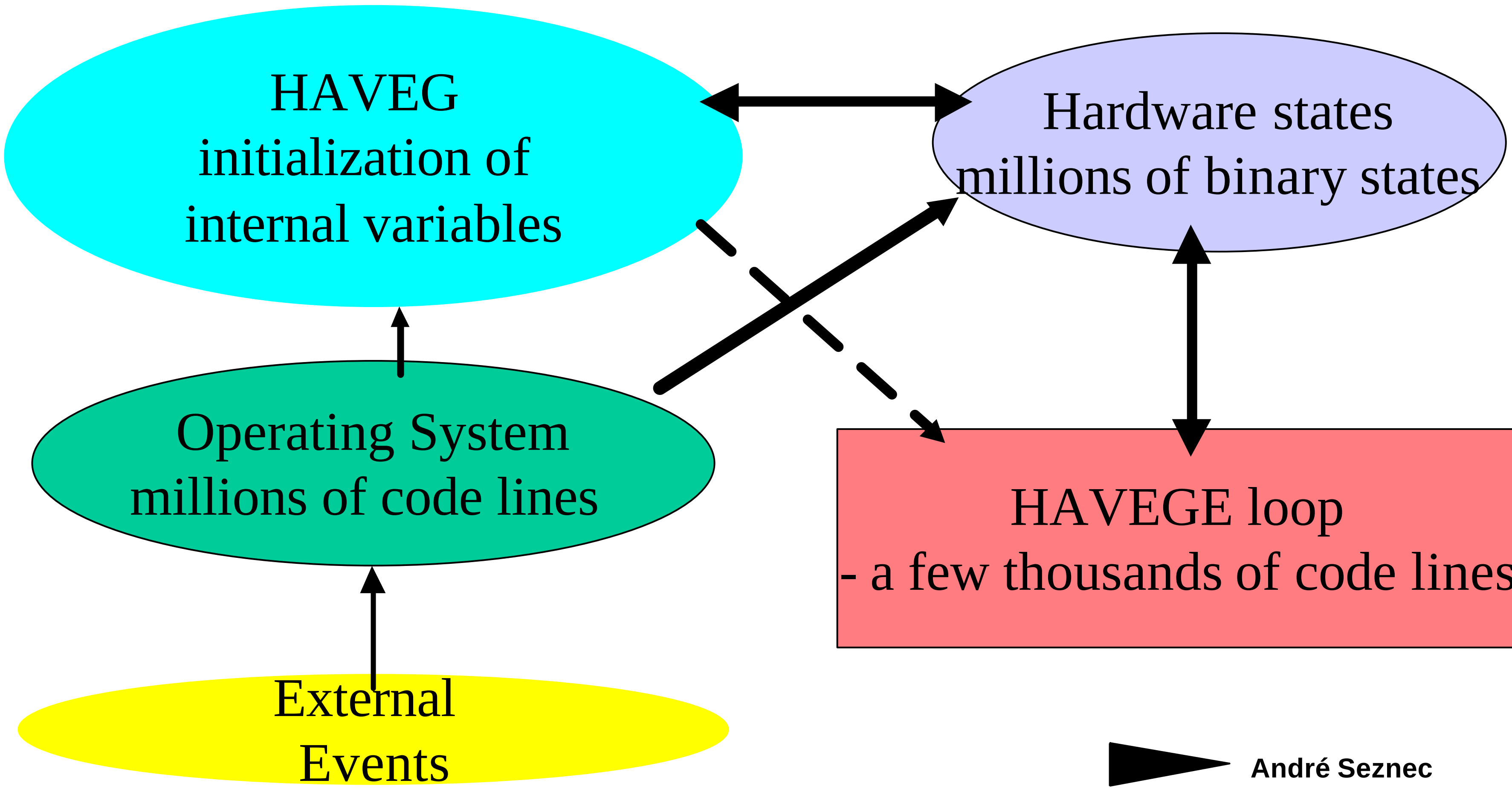
Faster and more uniformally distributed than random( )

Entropy Gathering + PRNG



André Seznec
Caps Team
Irisa

HAVEGE



André Seznec
Caps Team
Irisa

Further hiding of the internal state

HAVEGE sequences are unpredictable
but,

one may want to use other tricks to
further hide the internal state

Personalization

- On HAVEGE1.0 :
  - ➔ 1. random generation of parameters
    - constants, initialization, operators
  - ➔ 2. Recompilation
  - ➔ 3. At run time, the sequence depends on:
    - activity at run time
    - activity at installation time

Combining PRNGs with HAVEGE

- Yes, but I was really confident in my favorite PRNG
  - ➔ just embed your favorite PRNG in HardClock() :-)
  - ➔ and continuously reseed your second favorite with HAVEGE outputs !
- Reengineer HAVEGE with a robust PRNG:
  - ➔ take a robust PRNG code, add tests,unroll, etc to exercise hardware volatile states

Further possible tricks

- Use of a multithreaded HAVEGE generator:
  - ➔ share tables, pointers, code,
  - ➔ but no synchronization !!

- Use self-modifying code:
  - ➔ modify operators, constants on the fly with random values

Conclusion

- The interaction between user applications, external events, and the operating systems creates a lot of uncertainty in the hardware volatile states in microprocessor:
  - ➔ orders of magnitude larger than was previously captured by entropy gathering techniques.
- The hardware clock counter can be used at user level to gather (part of) this uncertainty:
  - ➔ HAVEG: a few 100 's Kbits/s
- PRNG and volatile entropy gathering can be combined:
  - ➔ HAVEGE: > 100 Mbits/s
    - unaccessible internal state
    - continuous and unmonitorable reseeding

Still not convinced ?

- Just test it:
  - ➔ http://www.irisa.fr/caps/projects/hipsor/HAVEGE.html
- Platforms:
  - ➔ UltraSparc II and III, Solaris
  - ➔ Pentium III, Pentium 4, Athlon - Windows, Linux
  - ➔ Itanium, Linux
  - ➔ PowerPC G4, MacOS 10
  - ➔ PocketPC