Concurrency as a Random Number Generator
Technical Report

Karolos Antoniadis Peva Blanchard Rachid Guerraoui Julien Stainer
École Polytechnique Fédérale de Lausanne
first.last@epfl.ch

Abstract
Concurrency is difficult to master because of the underlying non-determinism of shared memory accesses. In this paper, we present Co-RNG, a protocol that leverages this non-determinism to generate random numbers.

The power of Co-RNG comes from its simplicity. No specialized hardware is required: Two concurrent threads actively perform successive reads and writes to shared memory locations. Another thread collects the sequences of values read by these two threads and seeks to reconstruct the interleaving of read and write operations. The resulting (Markovian) interaction scheme is then used to produce random bits. This simplicity yields a transparent behaviour. If the hardware exhibits enough entropy, then Co-RNG efficiently extracts random numbers from it.

We successfully experimented Co-RNG on various idle as well as loaded platforms, from laptops and desktops equipped with Intel Core processors, to servers with Intel Xeon and AMD Opteron. Co-RNG passes all state-of-the-art random number generator statistical test suites while being faster than current I/O sampling based methods by 2 to 3 orders of magnitude.

1 Introduction
The demand for random numbers has become increasingly critical, especially in cryptographic applications. Basically, a random number generator must ensure that any adversary who observes any finite portion of the sequence of numbers produced should [11–13, 17]:

- see it as random (chaotic dynamics).
- not be able to guess the previously generated outputs (forward security).
- not be able to guess the next outputs (backward security or unpredictability).

The two first requirements are usually met by a (cryptographically secure) pseudo-random number generator (PRNG) [23], and are typically verified by passing a series of statistical tests, e.g., the NIST [29] (National Institute of Standards and Technology) and DieHarder [1] statistical test suites. Roughly speaking, a PRNG consists of a deterministic function successively applied to some initial state, namely the seed, to produce a sequence of numbers. A clever choice of the function yields chaotic dynamics, and if, in addition, the inverse of this function is hard to compute, then forward security is also ensured. However, unpredictability, which is critical in cryptographic applications, is obviously not guaranteed because an adversary that knows the current state of the PRNG can apply the deterministic function to compute the next outputs.

A true random number generator (TRNG) is expected to meet all three requirements. The behaviour of a TRNG is typically modeled by a sequence of independent identically-distributed unbiased random bits (Bernoulli process). The question of whether a TRNG really exists in the physical world is somehow philosophical, and has no definitive answer [7, 8, 10]. In practice, one usually looks for “empirically strong random numbers” [31]: some phenomena are believed to be truly random, because either no one understands their very nature (e.g., quantum-mechanical systems), or their detailed behaviour is so complex that the unavoidable errors in their measurements prevent any attempts to predict what will happen next (e.g., weather forecast, flow turbulence, thermal noise, etc.). One should remember though that this requires a leap of faith [10].

Interestingly, all (concrete) computers involve complex phenomena, and may contain, to a certain extent, sources of randomness, e.g., air turbulence effects in disk accesses [15], rotational latency of disks [22], hardware register content [31] or core-to-RAM latency [9]. The Linux kernel [19, 20] combines several sources such as

1 A cryptographically secure PRNG “only” ensures that (under unproven assumptions in complexity theory, e.g., existence of one-way functions), the adversary cannot learn the seed from the output of the PRNG. Yet, the adversary may use indirect means (e.g. knowledge of a flaw in the implementation of the seed generation) to get information on the seed, and thus on the current state of the PRNG.
delays between I/O interrupts, keyboard and mouse activity, network events, etc.

Having sources of randomness, however, is not sufficient. Once it is assumed that a source is random enough, the challenge is to extract actual random bits from it. The method based on air turbulence effects in disk accesses\(^\text{[15]}\) reaches a throughput of a few bits per minute. With the rotational latency of disks\(^\text{[22]}\), a throughput of hundreds of bits per minute is achieved. Nowadays, the I/O sampling based methods employed in the Linux kernel provide a throughput of a few bits per second.

For many cryptographic applications, such a throughput is not acceptable: generating one typical RSA key, for instance, usually requires thousands of random bits, which would take several minutes at least, using those methods. In practice\(^\text{[11]}\), a combination of several TRNGs are used to regularly reseed a cryptographically secure PRNG. This usually yields a higher throughput, since applying repeatedly the deterministic function underlying the PRNG to some seed executes faster. Whereas the PRNG guarantees chaotic dynamics and backward security, the periodic reseeding by TRNGs ensures unpredictability. However, if one is unsure about the quality of the seed (produced by a TRNG) then statistical attacks may become possible\(^\text{[20]}\). Besides, if the throughput of the TRNGs is not high enough, then either the PRNG blocks until a new seed is available, or continues at the price of longer predictable sequences in case of a compromised seed. For instance, the relay nodes of the Tor network consume a lot of random numbers as they are frequently generating keys in building circuits\(^\text{[18]}\); this is especially problematic as these nodes are often headless, and subject to a uniform workload, which greatly diminishes the throughput of current passive I/O-sampling based TRNGs. This issue is even more significant in cold boot of stateless systems such as the Linux Tails distribution\(^\text{[4]}\). Clearly, the need for faster TRNGs is crucial.

Some manufacturers have designed sophisticated dedicated hardware components\(^\text{[3, 24]}\) to provide random numbers at higher rates. Yet, these components are often expensive, not widely available, or not trusted by some users\(^\text{[3, 9, 31]}\), hence the motivation for combining various sources instead of relying on a single one\(^\text{[28]}\). In short, the fact that TRNGs are usually slow or rely on specific proprietary devices, has real consequences in concrete applications\(^\text{[28]}\).

The NIST has formulated specific recommendations on how to implement PRNGs\(^\text{[12]}\), TRNGs\(^\text{[13]}\) and how these should be combined\(^\text{[11]}\): a TRNG should not only guarantee a sufficient amount of random bits, but should also provide a transparent model of the underlying extraction process and operating conditions. Transparency here is the key to designing sufficiently discriminating criteria for diagnosing whether the underlying environment satisfies the assumptions required by the TRNG.

The challenge we address in this paper is the following: designing a randomness extractor from a hardware source that is fast (to maximize the unpredictability and reseed the PRNG sufficiently often), broadly available and transparent about its conditions of applicability.

The starting point of our approach is the very fact that some components of the hardware have indeed been recognized as hardly predictable\(^\text{[2, 15, 16]}\), e.g., memory traffic at the memory controller, jitter in latency of memory accesses, etc. Instead of directly measuring these phenomena (as in\(^\text{[25]}\)), however, we adopt an indirect approach by leveraging the presence of multiple cores on modern architectures through a simple idea that does not require any dedicated hardware. We let two concurrent threads perform successive reads and writes to shared memory locations. The schedule of their interleaving is a consequence of the micro-architectural phenomena at the level of the memory controller (which are not related to the OS task scheduler). Another thread collects the sequences of values read by these two threads and reconstructs the schedule of operations (up to indistinguishability). From this schedule, roughly speaking, we derive a Markov chain which we use as input for the Blum-Elias algorithm\(^\text{[33]}\), that acts as an extractor of entropy to finally produce the desired unbiased and independent random bits. Basically, the non-determinism of concurrency, usually considered a problem in devising reliable software, is exploited here to generate random numbers that could be used to secure applications.

Although the micro-architectural phenomena involved here have been recognized to be hardly predictable\(^\text{[14, 16]}\), we do not claim that these hardware phenomena involved are available in all configurations. We rather provide a way to extract random numbers from these phenomena whenever they occur in a configuration, as well as means for the user to assess the operability conditions of Co-RNG. In other words, we designed Co-RNG to be as transparent as possible by clearly separating the measurement (reconstructed interleaving of the shared memory accesses) from the entropy extraction (Blum-Elias). This is in sharp contrast with other implementations\(^\text{[6, 9, 31]}\) that do not clearly separate, the TRNG from the PRNG.

We have experimented with Co-RNG on several hardware platforms: from laptops and desktops equipped with Intel Core processors, to servers with Intel Xeon and AMD Opteron. We run our experiments on idle as well as on heavily loaded machines. We report on the throughput of Co-RNG and its evaluation by state-of-the-art RNG statistical test suites, namely, ENT\(^\text{[4]}\), NIST\(^\text{[29]}\) and DieHarder\(^\text{[1]}\). These experiments convey the fact that Co-RNG ensures higher throughput.
than usual passive I/O sampling methods by 2 to 3 orders of magnitude, while passing all the statistical test suites. Regarding transparency, we also propose several statistical criteria to assess whether the reconstructed interleaving of shared memory accesses satisfies the requirements of BLUM-ELIAS.

Our concrete implementation of Co-RNG takes the form of a (Linux) kernel module providing the reconstructed schedule of shared memory accesses, and a userspace daemon implementing the BLUM-ELIAS algorithm. The latter is parallelized to enhance the throughput. Last but not least, our deployment of Co-RNG ensures that an attacker cannot alter the output unless she has a privileged access to the system (in which case the system would be compromised beyond mitigation).

The rest of the paper is structured as follows. Section 2 overviews our Co-RNG protocol. Sections 3 to 5 detail the main components of Co-RNG. Section 6 presents the concrete deployment of Co-RNG, as well as the results of our experiments. Section 7 introduces simple and transparent criteria to assess the conditions of applicability of Co-RNG. Section 8 examines the security issues regarding a potential malicious attacker seeking to influence the output of Co-RNG. Section 9 finally discusses related work and concludes the paper.

2 Co-RNG: Overview

Our Co-RNG protocol involves two main sub-protocols, Co-Obs and Co-Rec, followed by a randomness extraction stage as seen in Figure 1. In this section, we give an overview of each of these parts. The details are given in Sections 3, 4, and 5 respectively.

2.1 Co-Obs

Co-Obs involves two atomic (linearizability [21]) is ensured by the use of memory barriers) single-reader-two-writer readers memory locations SM[0], SM[1], and two threads, $P_0$, $P_1$ performing a fixed number of concurrent successives rounds. The threads execute the rounds asynchronously, i.e., without waiting for each other. The number of rounds $N$ they execute is a Co-Obs parameter.

In each round, $P_i$ writes its current round number $n_i$ (starting from 0) into SM[$i$] (initialized to ⊥), and then reads the value from SM[1 − $i$] and stores it in obs[$i$]. The sequences obs$_0$, obs$_1$, hereafter called the observations of $P_0$, $P_1$, constitute the output of Co-Obs.

2.2 Co-Rec

After both threads $P_0$, $P_1$ finish their last round, another thread $Q$ collects their observations, and seeks to reconstruct what could have been the schedule of operations producing these observations. The difficulty for $Q$ is that the same sequence of observations by $P_0$ and $P_1$ could have been produced by different schedules. For example, if both threads $P_0$ and $P_1$ read the value 0 during their first round (obs$_0$[0] = obs$_1$[0] = 0), then $Q$ can only infer that the first write operations of $P_0$ and $P_1$ have been performed in parallel, but $Q$ is unable to decide whether the write operation of $P_0$ has actually happened long before, or slightly after, or at the exact same time as, etc. the write operation of $P_1$. In short, $Q$ only reconstructs the schedule to the precision allowed by the observations.

To capture these ambiguities, $Q$ computes a trace, given as a path in the reconstruction graph of Figure 2. In this graph, $w_i$ and $r_i$ denote respectively a write and a read by thread $P_i$. The symbol $[x || y]$ means that the operation $x$ is performed before the operation $y$. The symbol $[x | y]$, instead, means that the operations $x$ and $y$ are performed in parallel, and acknowledges the fact that $Q$ is intrinsically unable to give more precision about their ordering. In short, the computed trace sums up the possible schedules explaining the given observations.

2.3 Randomness Extractor

As it is, the trace computed by Co-Rec is an exact description of the schedule of operations up to indistinguishability. The starting point of Co-RNG is the fact that this trace may be seen as a source of randomness. The purpose of a randomness extractor is to convert such a trace into truly (i.e., independent and uniformly distributed) random bits. Many extractors have been developed and studied in the literature [26,30,32,33]. Roughly...
speaking, an extractor successfully extracts truly random bits from a weak source of randomness provided that the source satisfies some weak conditions. The first example, given by Von Neumann [26], extracts independent uniform random bits, from a sequence of independent biased bits. A more involved example is given by Santha and Vazirani [30] where several independent sources, individually partially controlled by an adversary, are combined to produce a quasi-random output. These examples are deterministic, while other approaches rely on an additional (small) truly random seed [27].

**Markov hypothesis.** Our choice of randomness extractor stems from the assumption that the interleaving of read and write operations is Markovian. The underlying intuition is that the choice of the scheduler for a given step only depends on a finite number of the immediately preceding steps (Markov property) and that this choice is not fully deterministic (non-degeneracy). We present tools to validate the Markov hypothesis in Section 4. Moreover we prove in [hidden link] that a non-degenerate Markovian scheduler yields non-degenerate Markovian traces after Co-REC.

**Blum-Elias.** The Blum-Elias extraction algorithm proposed in [33] is well-adapted to the above settings: Blum-Elias takes as input a trace of a Markov chain and outputs a sequence of independent unbiased random bits. One advantage of this algorithm is that it only needs to know the number of states of the Markov chain producing the trace. In particular, the values of the transition probabilities are not required. Moreover, Blum-Elias is deterministic in the sense that it does not require another short truly random seed.

Roughly speaking, Blum-Elias is a finite-state transducer: it analyzes a finite window in the trace, updates its state, outputs the corresponding bit string, and slides the window by one position. Note that the Blum-Elias algorithm is not a PRNG.

To the extent of our knowledge, there is no publicly available implementation of Blum-Elias, and thus, we had to implement our own version.

Note that, the choice of Blum-Elias as a randomness extractor relies mainly on the observation that the Markov hypothesis was not refuted in our concrete experiments. However, one is still free to choose another extractor since Co-REC always correctly rebuilds the schedule up to indistinguishability.

3 Concurrency Observation: Co-OBS

In this section, we detail the Co-OBS protocol.

**Protocol.** Co-OBS is the concurrency observation part of Co-RNG. It consists of two threads sharing two memory locations. Each thread successively writes one location to signal its progress and reads the other to observe the progress of the other thread.

The pseudo-code executed by each of the two threads is presented in Figure 3. Variable my_id ∈ {0, 1} denotes the identifier of the thread running the code, while her_id = 1 − my_id is the identifier of the other thread.

Both threads execute asynchronously N rounds as shown in Lines 2 to 8. During each round, a thread writes its current round number in the shared memory location associated to its identifier SM[my_id] (Line 4). It then reads the other shared memory location SM[her_id] (Line 6 and stores the read value in its private observation array obsmy_id, at the position corresponding to the current round number n. The two shared memory locations SM[my_id] and SM[her_id] are both initialized to the default value ⊥.

**Sequential interpretation.** It is important to notice that Co-REC needs the observations produced by the two concurrent threads to be consistent with at least one sequential ordering of their shared memory operations. Assume for example that each of the two threads executes N = 1 round. The three acceptable observations and their sequential explanations would be the following.

<table>
<thead>
<tr>
<th>obs0[0]</th>
<th>obs1[0]</th>
<th>Sequential explanations</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>0</td>
<td>([w_0 \cdot r_0][w_1 \cdot r_1])</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
<td>([w_1 \cdot r_1][w_0 \cdot r_0])</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>([w_0][w_1][r_0][r_1])</td>
</tr>
</tbody>
</table>

Recall that the notation \([w_0][w_1][r_0][r_1]\) actually captures the four indistinguishable possible sequences explaining the observations \((\text{obs}_0[0], \text{obs}_1[0]) = (0, 0)\). However, we want to avoid \((\text{obs}_0[0], \text{obs}_1[0]) = (⊥, ⊥)\), since these observations cannot be explained by any sequential ordering of shared memory operations.

**Memory ordering and caches.** To avoid such observations that have no sequential explanation, a memory barrier is inserted at Line 4. This barrier both prevents the pipeline of the processor from reordering the operations of a given thread and ensures that the value written in the shared memory at Line 3 is visible to other threads before the read operation of Line 6 starts. Notice that
we have also used compiler barriers to avoid any kind of reordering optimization. They are omitted from the pseudo-code of Figure 3 for the sake of clarity.

After each access to the shared memory, each thread flushes the corresponding cache line from its cache (Lines 6 and 7). This prevents deterministic cache effects by forcing both threads to write to and read from main memory. Finally, after both threads finish their N iterations, the two arrays obsi, i ∈ {0, 1} are passed on to Co-Rec for processing.

4 Schedule Reconstruction: Co-Rec

```
1: n₀ ← 0; n₁ ← 0
2: while n₀ < N ∧ n₁ < N do
3:   if obs₀[n₀] = ⊥ then
4:     trace[n₀ + n₁] ← [w₀ · r₀]
5:     n₀ ← n₀ + 1
6:   else if obs₁[n₁] = ⊥ then
7:     trace[n₀ + n₁] ← [w₁ · r₁]
8:     n₁ ← n₁ + 1
9:   else if obs₀[n₀] < n₁ then
10:  trace[n₀ + n₁] ← [w₀ · r₀]
11:  n₀ ← n₀ + 1
12:  else if obs₁[n₁] < n₀ then
13:  trace[n₀ + n₁] ← [w₁ · r₁]
14:  n₁ ← n₁ + 1
15:  else
16:  end if
17:  trace[n₀ + n₁] ← [w₀||w₁]
18:  while obs₁[n₁] > n₀ ∨ obs₀[n₀] > n₁ do
19:   if obs₁[n₁] > n₀ then
20:     trace[n₀ + n₁ + 1] ← [r₀ · w₀]
21:     n₀ ← n₀ + 1
22:   else
23:     trace[n₀ + n₁ + 1] ← [r₁ · w₁]
24:     n₁ ← n₁ + 1
25:   end if
26:  end while
27:  trace[n₀ + n₁ + 1] ← [r₀||r₁]
28:  n₀ ← n₀ + 1; n₁ ← n₁ + 1
29: end while
```

Figure 4: Co-Rec.

In this section, we give more details about the Co-Rec algorithm. The interested reader can read a full description in the companion technical report [hidden link].

Algorithm. The purpose of Co-Rec is to rebuild the schedule of the shared memory operations of the two threads of Co-OBS. Co-OBS ensures that there is at least one sequential order that is consistent with the observations it produces. But, as described in Section 2.2, some observations can be explained by several total orders on the shared memory operations of the two threads.

However, for each possible execution of Co-OBS for N rounds, the set of possible sequential explanations is exactly captured by a path of length 2 · N in the reconstruction graph of Figure 2. Note that this path starts necessarily in one of the three states on the left ([w₀||w₁], [w₀ · r₀] or [w₁ · r₁]) since both threads of Co-OBS start by writing (Line 1 of Figure 3). Reciprocally, assuming complete asynchrony, any valid path (starting in one of these states and containing N occurrences of each of the symbols w₀, w₁, r₀ and r₁) could theoretically be observed by Co-OBS.

The reconstruction algorithm underlying Co-Rec is presented in Figure 4. The rebuilt path is stored in the array denoted by trace[0, . . . , 2 · N − 1]. The variable n₀ (resp. n₁) stores the number of rounds that have been rebuilt so far for P₀ (resp. P₁). Accordingly, n₀ (resp. n₁) is also the number of the next round to rebuild for P₀ (resp. P₁). The length of the rebuilt path in the reconstruction graph of Figure 2 is consequently n₀ + n₁, i.e., the total number of rounds rebuilt.

The reconstruction loop (Lines 2 to 29) runs until N rounds are rebuilt for one of the two threads. Each iteration of the reconstruction loop always rebuilds an integral number of rounds, i.e., no half-rounds, for each thread. The algorithm explores all cases: those when both processes are about to write Lines 3 to 16 and those when they are about to read Lines 17 to 27.

After the whole schedule is rebuilt, the potential solo runs at the beginning and end of the trace (repetitions of one of the states [w₀ · r₀] or [w₁ · r₁]) are discarded and the trace is passed to Blum-Elias that uses it to extract a random binary sequence. In practice, these solo runs are essentially due to the fact that the threads may not start exactly at the same time and may progress at slightly different speeds. Note, that the algorithm in Figure 4 could potentially generate a path with a length shorter than 2 · N due to the loop condition in Line 19. This is acceptable since the missing steps correspond to the solo run of the last process and are discarded anyway.

5 Randomness Extractor: Blum-Elias

In this section, we give a detailed presentation of our implementation of the Blum-Elias algorithm. The original algorithm has been proposed in [33] to which the interested reader can refer for further details.

Algorithm. This algorithm takes as an input the trace of a Markov chain and outputs a sequence of unbiased independent random bits. Blum-Elias only needs to know the number S of states of the Markov chain; the transition probabilities remain unknown. We illustrate
the procedure for our case in Figure 5, assuming the transitions of the Markov chain follow the reconstruction graph in Figure 2. The algorithm parses the trace, reading one state after another, and maintains a set of windows associated with each possible Markov state. At each step, if the previous state is, e.g., \([w_0][w_1]\), and the state being read is \([r_0][r_1]\), then \([r_0][r_1]\) is appended to the window associated with \([w_0][w_1]\). Moreover, if the state \([w_0][w_1]\) is reached again, and the associated window has reached a predefined maximum size, then a specific function \(f_E\), called the Elias function, is applied to the window to produce a finite binary word. This word is appended to the output, the window is flushed, and the algorithm continues.

![Figure 5: BLUM-ELIAS details.](image)

**Parallelism.** The Elias function \(f_E\) is polynomial-time in the size of the window, and forms the main computation cost in BLUM-ELIAS. Our implementation allows to distribute this cost among many threads to speed up the computation. To do so, the whole trace is divided into as many blocks as there are threads. Each thread holds an instance of BLUM-ELIAS to be executed on its block. One must be careful though since the initial state (i.e., the content of the windows) of one BLUM-ELIAS instance depends on the final state of the preceding instance. Fortunately, by parsing the whole trace once without computing the Elias function, we can compute the initial state of each instance quite quickly. Then, each thread executes the rightly initialized BLUM-ELIAS instance. Their outputs are orderly concatenated to form the final output.

**Markov lag.** So far, we have assumed that the Markov chain fed to BLUM-ELIAS is directly represented by the reconstruction graph of Figure 2. This amounts to assume that the Markov chain has lag 1, i.e., that the probability distribution of the next symbol to schedule is entirely determined by the last symbol that has been chosen. However, 

\[ \text{probability distribution} \]

...This amounts to assume that the Markov chain has lag 1, i.e., that the probability distribution of the next symbol to schedule is entirely determined by the last symbol that has been chosen. However, a priori, the probability distribution of the next symbol may be determined by more than one of the last chosen symbols (lag \(l > 1\)). To reduce to the case of lag 1, we perform the following transformation. Given a trace \(T = X_0X_1 \ldots\), we compute a new trace \(T' = Y_0Y_1 \ldots\), where \(Y_i = [X_i \ldots X_{i+l-1}]\) is a symbol encoding the segment of length \(l\) starting at position \(i\) in the original trace. It is easy to see that \((Y_i)_{i \in \mathbb{N}}\) is a Markov chain of lag 1, with \(6 \cdot 3^{l-1}\) states. Since BLUM-ELIAS only needs to know the number of states, we can easily apply this algorithm to the trace \(T'\).

This transformation is rather harsh, since the number of states \(6 \cdot 3^{l-1}\) is exponential in the lag \(l\). This impacts the computation cost of BLUM-ELIAS. There is room for optimization though, because many of these states are redundant. One could derive a lighter Markov model (via Markov chain reduction techniques) to convert our trace to, and reduce the computation cost.

**6 Experimental Evaluation**

In this section we present quantitative and qualitative tests of Co-RNG. We first describe our experimental setting, and then present the throughput measurements as well as the evaluation of the quality of our generated sequences by state-of-the-art statistical test suites.

**Implementation.** Co-RNG is implemented for Linux in C and compiled with the GNU Compiler Collection (GCC). The sub-protocols Co-ObS and Co-Rec are implemented as a kernel module, named coobs-mod, that exports the character device file `/dev/coobs`. This character device provides traces on demand: reading \(N\) bytes from `/dev/coobs` launches the Co-ObS protocol, with the two threads racing for \(N\) rounds\(^3\) and then executes the Co-Rec protocol. Moreover, each of the two Co-ObS threads reserves a core for itself (disabling preemption from the OS scheduler) before executing Co-ObS. The cores are released right after the race.

The randomness extractor BLUM-ELIAS is implemented as a user-space program, `corng-daemon`. This program has two modes, a testing and a daemon mode. In testing mode, `corng-daemon` takes as a parameter the number of requested random bytes. It then retrieves a trace from `/dev/coobs` that is subsequently processed according to the BLUM-ELIAS algorithm. If the generated output is long enough compared to the trace length,

\[ \text{We store two successive symbols of the trace into one byte. In particular, the number of rounds executed by each process equals the requested number } N \text{ of bytes.} \]

---

\[ ^3 \text{We store two successive symbols of the trace into one byte. In particular, the number of rounds executed by each process equals the requested number } N \text{ of bytes.} \]
the program writes the result to the standard output, and proceeds by reading another trace. Otherwise, the program discards the trace, and reads another trace. The program repeats this until the requested number of random bytes has been produced. This discarding mechanism is used to prevent spurious results of Blum-Elias due to lock-step traces \((w_0|w_1|w_2|w_3|w_4|w_5|w_6|w_7|\ldots)\). We give more details about this issue in the Robustness paragraph that follows later on.

In daemon mode, corng-daemon is registered as a kernel entropy source so that, whenever the entropy pool is almost empty, the kernel asks corng-daemon to provide it with random bytes. Those random bytes are produced in the same way as in testing mode.

As said before, there was no available implementation of Blum-Elias. Providing one was not trivial as it required a careful implementation of sophisticated combinatorial computations (e.g., recursive combination of multinomial coefficients for ranking functions [33]). Our code uses the GNU Multiple Precision Arithmetic Library libgmp for computations on large integers and is parallelized using the GNU implementation of OPENMP. The parameters are the Markov lag and the number of threads to use for the computation.

For security reasons, the character device file /dev/coobs can only be accessed by the root user, and corng-daemon runs with root privilege.

We chose to separate the trace generation (Co-OBS, Co-REC) and randomness extraction (Blum-Elias) for the sake of transparency (the traces can be more easily audited) and modularity (the randomness extractor can be more easily modified or replaced, if need be).

The module coobs-mod (Co-OBS, Co-REC) and the program corng-daemon (Blum-Elias) comprise about 370 and 300 lines of C code respectively. The source code of Co-RNG is available at [hidden link].

Platforms. We conducted our experiments on the following hardware platforms, all running GNU/Linux.

**Arch 1.** Dual-socket Intel Xeon E5-2680 v2 @ 2.8 GHz with 10 cores per socket and 2 hardware threads per core (total of 40 virtual cores).

**Arch 2.** AMD Opteron 6162 @ 2.1 GHz featuring 8 sockets of 6 cores (total of 48 cores).

**Arch 3.** Quad-socket Intel Xeon E7-4830 v3 @ 2.1 GHz with 12 cores per socket and 2 hardware threads per core (total of 96 virtual cores).

**Arch 4.** Intel Core i5-2520M @ 2.5 GHz with 2 cores and 2 hardware threads per core (total of 4 virtual cores).

**Arch 5.** Intel Core i5-4278U @ 2.6 GHz with 2 cores and 2 hardware threads per core (total of 4 virtual cores).

**Arch 6.** Intel Core i7-4650U @ 1.7 GHz with 2 cores and 2 hardware threads per core (total of 4 virtual cores).

**Arch 7.** Intel Core i7-4800MQ @ 2.7 GHz with 4 cores and 2 hardware threads per core (total of 8 virtual cores).

**Deployment and parameters.** We describe the parameters that gave the best results for each platform. For all the platforms, except for the Opteron (Arch 2), the two Co-OBS threads were pinned to the same physical core, running as hyperthreads, and the shared memory locations belong to the same cache line. As explained above, for all machines, the cache lines are flushed after each shared memory access.

On the NUMA platforms (Arch 1, 2 and 3), the shared memory locations used by Co-OBS are allocated in a specific memory node. On the two Xeon’s (Arch 1 and 3), the shared memory locations are attached to the NUMA node where the Co-OBS threads run.

The case of the Opteron (Arch 2) is slightly more involved. Figure 6 depicts the topology of the inter-socket connections on this machine. Each box represents a NUMA node hosting six cores and a memory module.

The two Co-OBS threads are pinned to the first core of NUMA nodes N0 and N6, while the shared memory locations are allocated on the memory module of node N5. Thus, each thread is 2-hops away from the Co-OBS shared memory locations. This increases the variability of the memory accesses latencies. The shared memory locations in Co-OBS lie in two distinct cache lines.

The Blum-Elias algorithm runs with window size 20, lag 4 for the Opteron (Arch 2), and lag 3 for all the other platforms.

**Throughput evaluation.** The daemon program is set up to produce 128 Mbits of random data. To do so, the daemon reads traces of fixed length from /dev/coobs as many times as required. The throughput corresponds then to 128 Mbits divided by the elapsed time. Table 1 reports the measured values for each architecture. The column sequential (resp. parallel) reports the throughput when Blum-Elias runs on a single core (resp. all available cores).

As a reference, the table also displays the throughput of HAVEG (the TRNG underlying HAVEGE) as

---

**Figure 6: Inter-socket connections on the Opteron.**

Arch 6, Intel Core i7-4650U @ 1.7 GHz with 2 cores and 2 hardware threads per core (total of 4 virtual cores).

Arch 7, Intel Core i7-4800MQ @ 2.7 GHz with 4 cores and 2 hardware threads per core (total of 8 virtual cores).
Statistical tests. For each platform, we have generated 20 sequences of 16 MB of random data. Figure 9 displays the entropy of these sequences as estimated by the ENT test suite [2]. The entropy is measured in number of bits per byte of data. Thus, the purer the randomness, the closer the estimation is to 8.

Figure 7 displays the results for the NIST statistical tests.
test suite [29] (Figures 7a, 7b and 7c) and DieHarder [1] (Figure 7d). For each sequence in our dataset, the NIST suite performs a series of 188 tests on $N$ samples of the sequence. The samples are consecutive $S$-bits segments of the original sequence. Figures 7a to 7c correspond to three choices for the couple $(N, S)$. In each figure, for each architecture, for each $0 \leq f \leq 7$, we plot the number of sequences (among the 20) which failed exactly $f$ tests. In all cases, no sequences failed more than 6 tests.

The results of DieHarder are displayed in Figure 7d. Each test in DieHarder is qualified as passed, weak or failed. For all our sequences, no tests were qualified as failed. Therefore, in Figure 7d, we plot the data as in the case of the NIST, except that $f$ is the number of weak tests. There were never more than 7 weak tests.

As a baseline, we display the corresponding results for the cryptographically secure pseudo-random number generator provided by /dev/urandom. From these results, we conclude that these tests do not reject the fact that Co-RNG provides high quality random numbers.

**Robustness.** The results depicted in Figure 7 were performed on idle machines. Here, we show that running Co-RNG on a loaded machine does not alter the quality of the output. The experiment consists in running Co-RNG while repeatedly compiling a Linux kernel (using all the cores available). The experiments were performed on Arch. 4 and Arch. 7, as they are sufficiently small to be conveniently fully loaded. For each of them, 20 sequences of 16 MB were generated. Figure 8 displays the results of the NIST and DieHarder statistical test suites. As a baseline, we display (again) the results for /dev/urandom. These results confirm that our implementation is resilient to blind flooding of the system.

This robustness is ensured by the discarding mechanism. Indeed, we observed experimentally that either the traces looked as in the idle situation, or exhibit a lock-
step behaviour \( (|w_0|w_1][r_0]|r_1][w_0]|w_1) \ldots \). However, in the latter case, BLUM-ELIAS output is very small compared to the length of the trace, and our implementation discards the trace. The percentage of discarded traces in the experiments on the loaded machines ranges from 10\% to 40\% on average. (On the idle machines, this percentage is less than 1\%).

7 Markov Hypothesis Validation

CO-REC is always able to rebuild the schedule (up to indistinguishability) of the two threads of CO-OBS, while BLUM-ELIAS needs to satisfy two additional requirements for its output to be truly random: the trace output by CO-REC has to satisfy the Markov property, and be non-degenerate. In particular, checking the conditions of applicability of CO-RNG boils down to verifying that the output of CO-REC satisfies the Markov property and that it is non-degenerate. In this section, we give means to assess the validity of these assumptions, as well as evidence supporting the hypothesis in the experimental settings presented above.

Markov property. Recall that a stochastic process \((X_t)_{t \in \mathbb{N}}\) satisfies the Markov property if for some constant \(l\), namely the lag, for all \( t \geq l \), \( P(X_t | X_{t-1}, \ldots, X_0) = P(X_t | X_{t-l}, \ldots, X_{t-1}) \).

To test this hypothesis, we proceed as follows. We fix an arbitrary path \( \pi \) (in Figure [2]) of length the lag \( l \). We then compare, for each state \( x \) before \( \pi \), the conditional probability distribution \( P(z | x \pi) \) with the distribution \( P(z | \pi) \), where \( z \) runs over the states out of \( \pi \). More precisely, we define \( \sigma_x^\pi \) as the \( l^2 \)-distance between the distributions \( P(\cdot | x \pi) \) and \( P(\cdot | \pi) \). Intuitively, if the trace has the Markov property with lag \( l \), the deviation \( \sigma_x^\pi \) should be small for all patterns \( x \pi \) of length \( l + 1 \). To visualize this measurement, we plot a survival function: for each value \( s \) on the \( x \)-axis, we report, on the \( y \)-axis, the percentage of pattern \( x \pi \) of length \( l + 1 \) such that \( \sigma_x^\pi \geq s \). Therefore, the more the curve is concentrated around \( s \approx 0 \), the more the trace appears to be Markovian with lag \( l \).

Figure [10] displays the survival functions for i7-4800MQ (Arch. 7) with lag \( l = 1, 2, 3, 4 \) respectively. We see that, indeed, the curves are more and more concentrated around 0. Figure [11] displays the survival functions for the same architecture, when the flushing of the cache lines is disabled. We see that the curve does not concentrate around 0 for lag values below 4.

Non-degeneracy. Another property one has to check is the non-degeneracy of the Markov chain. Indeed, if there are too many states \( x \) with a transition \( x \rightarrow y \) having very high probability, i.e., an “almost deterministic” Markov chain, then there is not enough entropy in the process. To measure the degeneracy, we compute for each path \( \pi \) of length the lag \( l \), the outward (min) entropy \( h_\pi \) of \( \pi \), defined by \( h_\pi = -\min \sum P(z|\pi) \log P(z|\pi) \). For a more compact information, we can also compute the average (min) entropy \( \mathcal{H} = \sum \mu(\pi) h_\pi \), where \( \mu \) is the stationary distribution.

To visualize this data, we proceed as follows. For each pattern \( \pi \), we plot a dot with coordinates \((\mu(\pi), h_\pi)\). Figures [12] and [13] present the results on traces produced by CO-OBS on the i7-4800MQ (Arch. 7), respectively with and without the flushing of cache lines enabled. In both cases, lags from 1 to 4 are displayed. Note that since there are only 3 possible transitions out of any pattern, the maximal value of the outward entropy is \( \log 3 \approx 1.585 \) bits. Note also that the outward entropy
Figure 12: Entropy diagram (Arch. 7). Stationary probability of pattern (x-axis), outward entropy (y-axis).

Figure 13: Entropy diagram (Arch. 7 - cache line flush disabled). Stationary probability of pattern (x-axis), outward entropy (y-axis).

8 Security Considerations

In this section, we analyze the possible security threats against Co-RNG and how they are prevented. Recall that in our design, Co-ObS and Co-Rec are implemented as a kernel module servicing the character device file /dev/coobs, while Blum-Elias is implemented as a daemon which reads from /dev/coobs to feed the Linux entropy pool. We recommend executing the daemon with root privilege, and restricting the access rights of the device to the root user only.

Of course, if the attacker already has root privileges or direct control of the OS kernel then she can access (and modify) every element from the system memory, entropy pool, PRNG internal state and output, etc., so the system is compromised beyond mitigation. We thus assume that the attacker is unprivileged.

Since Co-ObS, Co-Rec and Blum-Elias are run as privileged processes, the attacker cannot read nor modify their memory. If the attacker has only access to the output of the PRNG (/dev/random or /dev/urandom), then, by definition of a cryptographically secure PRNG, she cannot discover the internal state nor the content of the entropy pool. Therefore, the only means for the attacker to hinder the system, is to influence what Co-RNG injects in the entropy pool. Since the attacker has no access to the memory of the Co-RNG threads, she can only try to manipulate the interleaving of Co-ObS threads to get insights about the resulting output of Co-RNG (thus of the PRNG). Note, that even if a non-privileged user is able to see the effects of the memory operations in the shared last level cache, the attacker still has no access to the contents of the cache line, and cannot know which of the two threads of Co-ObS is responsi-
ble for a given operation. It is consequently not providing her with any insight on the current interleaving.

Remember that in the kernel module, in the two cores where the kernel threads execute CO-OBS, preemption is disabled. This prevents an attacker from exploiting the OS thread scheduling policy, e.g., pinning parasitic processes on the same cores. Furthermore, the number of rounds can be adjusted such that the two cores are isolated only for a short period of time.

Hence, the attacker’s last possibility is to influence the interleaving of memory accesses at the level of the memory controller. For the attacker to succeed, this strategy requires to know both the inner workings of the memory controller, as well as the current set of running processes and the timing of their memory accesses. We trust that, from any practical point of view, an unprivileged attacker cannot access such information. Moreover, from our experiments, a naive blind flooding of the memory controller only makes CO-OBS generate traces that are either undisturbed, or lock-step, in which case the daemon easily discards these traces. In other words, such an approach can only affect the throughput of CO-RNG.

9 Concluding Remarks

We presented CO-RNG, a tool to extract random numbers from the interleaving of concurrent memory accesses. CO-RNG involves three stages: CO-OBS that lets two threads perform a series of concurrent read and write operations to the shared memory, CO-REC that collects the values read by the threads of CO-OBS to build a trace, i.e., a reconstruction of a sequential ordering of their memory operations (up to indistinguishability), and finally, BLUM-ELIAS which is our implementation of the algorithm proposed in [33], processing the output of CO-REC to produce a sequence of bits.

The schedule observation and reconstruction, namely CO-OBS and CO-REC, do not rely on any assumption about the underlying scheduler. CO-REC always produces an exact description of the set of possible sequential explanations for the observation output by CO-OBS. In contrast, to produce truly random bits, BLUM-ELIAS requires the trace produced by CO-REC to be a realization of a non-degenerate Markovian chain. We proved in a companion technical report [hidden link] that when the scheduler is a non-degenerate Markovian process, the trace computed by CO-REC is also a non-degenerate Markovian process. Thus, in such a theoretical model, CO-RNG does produce truly random bits.

One of the main advantages of our implementation is transparency. Thanks to the separation of the generation of the trace (CO-OBS and CO-REC) from the randomness extraction (BLUM-ELIAS), the user can easily audit the quality of the traces, and assess the operability conditions of CO-RNG. We presented simple and transparent statistical criteria to do so. We have shown that, for several platforms and configurations, these criteria provided evidence supporting the non-degenerate Markovian nature of the scheduler.

Note moreover that we do not claim that the scheduler is Markovian and non-degenerate in all configurations. The purpose of CO-RNG is to extract randomness whenever the configuration exhibits enough entropy. Also, thanks to our discarding mechanism, our implementation also avoids outputting spurious results when the scheduling of the threads is degenerate.

We evaluated the throughput and the quality of the output of CO-RNG on several platforms. The experiments show that CO-RNG reaches a throughput greater than classic I/O sampling methods by 2 to 3 orders of magnitude on idle machines, while passing the state-of-the-art statistical test suites [1][29]. We also showed that our implementation was robust in case of loaded machines, in the sense that the output of CO-RNG still passed the test suites.

Other attempts have tried to leverage uncertainty in hardware microstates, such as the CPU Jitter RNG [25]. CPU Jitter leverages the jitter that exists in underlying hardware mechanisms, e.g., CPU execution times, in order to generate random numbers. In contrast to CPU Jitter, CO-RNG relies on a mathematically proven randomness extractor and allows to check that the observed phenomenon meets the requirements of this extractor. Moreover, the throughput of CPU Jitter is less than ours by 1 to 2 orders of magnitude.

HAVEGE [31], on the other hand, exploits the unpredictable state of a core after hardware interrupts to seed a specific kind of random walk. This random walk is itself used to produce random numbers. In this case, the TRNG part, called HAVEGE, comprises the unpredictable core state after interrupts, whereas the PRNG part consists in the random walk. The throughput of HAVEGE is about 100 Kbit/s, and that of the PRNG part is about 100 Mbit/s. Unfortunately, the two parts are intertwined to the extent that it is difficult for the user to audit the output of the TRNG part. This is arguably one of the main drawbacks of the approach of HAVEGE: the behaviour of the underlying generating process and the reliability of the random walk are unclear.

Similarly, hardware implementations, such as the Intel instruction RD_RAND, often embeds both the TRNG and the PRNG into a black-box component which makes it cumbersome for the user to evaluate each of them. According to [9], the TRNG part behind RD_RAND relies on thermal noise in a self-timed circuit, and claims to reach a throughput of 3 Gbit/s. The generated stream then feeds an AES-based PRNG to produce the final random bits. One may wonder though why the hardware instruction RD_RAND requires a PRNG after the TRNG
if the latter claims to reach a throughput of 3 Gbit/s. Possibly, the quality of the output of the TRNG is not good enough, which is difficult to assess given the black-boxing of the components.\footnote{Many developers have voiced their concern about the extent to which one can trust RD_RAND\cite{5}.}

In addition, Co-RNG presents additional interesting properties. The algorithm is potentially deployable on any multi-core architecture. Contrarily to HAVEGE\cite{31} or passive I/O sampling techniques, Co-RNG is active, and does not rely on system or user events happening on the machine which may not be frequent enough, for example on headless machines under uniform workloads. Finally, we have explained how the case of a malicious attack able to influence the output of Co-RNG boils down to having a privileged access to the system.

References


[2] ENT - A pseudorandom number sequence test program. \url{http://fourmilab.ch/random}

[3] ID Quantique. \url{http://www.idquantique.com}


