

An efficient high quality random number generator for multi-programmed systems

André Zúquete

IEETA/UA, Campus Univ. de Santiago, 3810-193 Aveiro, Portugal
Tel.: +351 234 370504; Fax: +351 234 370545; E-mail: avz@det.ua.pt

This document presents an efficient, high quality random number generator for multi-programmed environments, in particular for UNIX/Linux and Windows systems. The algorithm uses a system's notion of the time, given by a high-precision real-time counter, to obtain random bits, and a combination of system calls to reduce the correlation between consecutive random bits. The combination of system calls introduces variable and unpredictable real-time gaps in the execution of the generator. We present a rationale for such variation and unpredictability, and we show that in fact they exist even when the generator runs in an adverse scenario, i.e., a lightly loaded system. We also show how the generator resists several attacks attempting to guess or control the values it produces. The quality of the generator is evaluated both in terms of its performance and the randomness of the byte sequences it produces. Comparing against other similar generators, `CryptoLib` and `librand`, our generator produces equally good random byte sequences, but its performance increases with the speed of the processor, while the performance of `CryptoLib` and `librand` is rather constant for each operating system, independently of the processor were it runs.

Keywords: Random number generators, operating systems, Pentium processors, time samples

1. Introduction

The generation of true random sequences of numbers is very useful for many applications, like cryptographic applications [20,25] or Monte Carlo simulations [14]. Their major application in cryptography is for generating secret random data, like cipher keys. User-level cryptographic applications, like PGP [22] or SSH [28], key distribution servers, like Kerberos [29], and most protocols for supporting secure communications [7,15] need to use truly random numbers to generate public-private key pairs, temporary cipher keys, challenge/response nonces (or cookies), etc.

True random number generators are difficult to implement, being often replaced by cryptographically strong pseudo-random generators seeded with truly random seeds [13,25]. These pseudo-random generators are risky if attackers could derive a limited search space for guessing the seeds [9] and most software techniques to generate random sequences or seeds have known problems [8]. Hardware techniques are an alternative to overcome the pitfalls of pure software techniques. However, hardware techniques force systems to use secure and dedicated devices capable of producing some sort of random noise [1,5], which may be expensive or not available.

There are some alternative solutions for generating true random sequences without specific hardware devices. One solution is to rely on user-provided input (passwords, time between key strokes, mouse motion, etc.) but that is burdensome for users and is inadequate for servers and communication protocols. Another solution is to feed a randomness pool with random data gathered from unpredictable environment events (users' input activities, hardware interrupts, arrival of network packets, etc.) and a cryptographically strong function to distil random bits from that pool, but the quality of the generator depends on the frequency of the input events and the security of the randomness pool is an issue [13].

Hence, for the particular case of cryptographic applications, which are becoming more and more widespread, there is a real need for secure random number generators, not relying on user-provided input or unpredictable environment events, capable of producing high quality random sequences at high speed, depending only of the running platform – hardware support and operating system.

In this document we present a practical and efficient software algorithm for producing high quality random byte sequences in most multi-programmed systems – single-user computing environments (personal computers) or multi-user computing environments (workstations). The algorithm is software based, i.e., it does not use any hardware techniques directly, and may be implemented in most systems running UNIX, Linux and Windows. In terms of hardware support, the algorithm only needs an high-precision hardware real-time timer, that is available in most personal computers and workstations. In this work we used the Time-Stamp Counter register of the Intel Pentium processor, which keeps an accurate 64-bit count of clock cycles that occur in the CPU.

Each random bit is extracted from the current system time, and the extraction of consecutive random bits is separated by random real-time gaps much higher than the real-time clock resolution. If the real-time clock resolution is close to the time expended by each machine instruction, then random real-time gaps can be created by executing complex system operations. In fact, the time expended by system operations depends on many factors, either related to the operating system (search algorithms, allocation and deallocation of resources) or related to the CPU execution (caching, interrupt handling). Complex system operations that have multiple alternative execution paths, or access a large amount of system information, are very sensitive to all those factors, yielding a random real-time execution time. Thus, for multi-programmed systems we can use the time expended by process (or thread) creation and termination cycles as a source of random real-time gaps. In our implementation for UNIX/Linux and Windows systems we create and wait for the immediate termination of a new process/thread in order to introduce a random real-time gap in the execution of the random generator.

The randomness of the byte sequences produced by the generator was evaluated with several statistical tests: the frequency test, the serial test, the poker test, the binary derivative test, the runs test, and the universal test [3,4,6,10,11,18,21]. The generator was implemented and fully evaluated in recent versions of the two most popular operating systems (Linux and Windows) running on personal computers based

on the Pentium processor (Pentium II at 233 MHz and 331 MHz and Pentium III at 1 GHz). This evaluation showed that our generator produces good random byte sequences, similar in statistical quality to the ones generated by two similar generators – the true random byte generators of `CryptoLib` and `librand` – but is much faster than these generators. Furthermore, its performance grows with the speed of the processor, while the performance of the other two generators is rather constant for each operating system, independently of the processor where it runs.

The rest of the paper is structured as follows. The next section presents related work. Section 3 describes the algorithm of the random byte generator. Section 4 presents the rationale for the source of entropy used by the generator, and presents an analysis of the effectiveness and security of such source. Section 5 evaluates the randomness quality of the sequences produced by the generator. Section 6 evaluates the performance of the generator. Section 7 discusses the usage of our generator in other systems. Finally, Section 8 concludes this document.

2. Related work

There are many techniques to generate good random number sequences [24,25]. Some techniques rely on user-provided input [23], being only suitable for interactive applications; others use hardware devices [1,5], increasing the cost and complexity of the support systems; others use strong pseudo-random generators [25], which do not produce truly random sequences, fed with some random data gathered from the execution environment of the generator [13]. As we want to provide a true random number generator for a broad range of systems, without special hardware devices, or applications, we will not address those techniques here. We will mainly describe two techniques that, like our algorithm, use the uncertainty of the operating system behavior to compute random numbers. These techniques are implemented by the libraries `CryptoLib` [16] and `librand`¹. To conclude this section we will also describe another generator, the one implemented by the Linux kernel, that is also purely software based but uses a different approach for gathering random data – it waits for random events, instead of forcing them to appear, and we briefly compare it with a similar one, designed to run at user-level and in many systems [13].

2.1. Generators of `CryptoLib` and `librand`

`CryptoLib` is a library for software cryptography that includes a true random generator. The algorithm of this generator, for UNIX systems, uses signals raised by the real-time timer (SIGALRM) to stop incrementing an application counter, and mixes 11 times the value of this counter to generate a 32-bit value. Signals are programmed to be raised 16 665 μ s after the start of counting. Each random byte is the low-order byte resulting from hashing, with the Secure Hash Algorithm (SHA [26]), each of these 32-bit values (see Fig. 1).

¹Publicly available at <ftp://ftp.research.att.com/dist/mab/librand.shar>.

CryptoLib	librand
<pre> static jmp_buf env; static unsigned count; static unsigned ocount; static unsigned buffer; static int tick() { struct itimerval it, oit; timerclear(&it.it_interval); it.it_value.tv_sec = 0; it.it_value.tv_usec = 16665; if (setitimer(ITIMER_REAL, &it, &oit) < 0) perror("tick"); } static void interrupt() { sigsetmask (0); if (count) longjmp(env, 1); signal(SIGALRM, interrupt); tick(); } static unsigned long roulette() { if (setjmp(env)) { count ^= (count>>3) ^ (count>>6) ^ ocount; count &= 0x7; ocount=count; buffer = (buffer<<3) ^ count; return buffer; } signal(SIGALRM, interrupt); count = 0; tick(); for (;;) count++; } unsigned long truerand() { count=0; roulette(); roulette(); roulette(); roulette(); roulette(); roulette(); roulette(); roulette(); roulette(); roulette(); return roulette(); } void reallyRandomBytes(unsigned char *buf, int numbytes) { register int i; register unsigned char *bp; Ulong a, *aa; i = numbytes; bp = buf; for (i=0; i<numbytes; i++) { a = truerand(); aa = shs((unsigned char *)&a, 4); bp[i] = (unsigned char)aa[0]; } } </pre>	<pre> static jmp_buf env; static unsigned count; static unsigned ocount; static unsigned buffer; static int tick() { struct itimerval it; timerclear(&it.it_interval); it.it_value.tv_sec = 0; it.it_value.tv_usec = 16665; if (setitimer(ITIMER_REAL, &it, NULL) < 0) perror("tick"); } static void interrupt() { sigsetmask (0); if (count) longjmp(env, 1); signal(SIGALRM, interrupt); tick(); } static unsigned long roulette() { if (setjmp(env)) return count; signal(SIGALRM, interrupt); count = 0; tick(); for (;;) count++; } unsigned long raw_truerand() { void (*oldalarm)(); struct itimerval it; unsigned long counts[12]; unsigned char *qshs(); unsigned char *r; unsigned long buf; int i; getitimer(ITIMER_REAL, &it); oldalarm = signal(SIGALRM, SIG_IGN); for (i=0; i<12; i++) { counts[i]=0; while ((counts[i] += roulette()) < 512); } signal(SIGALRM, oldalarm); setitimer(ITIMER_REAL, &it, NULL); r = qshs(counts, sizeof(counts)); buf = *((unsigned long *) r); return buf; } unsigned long randbyte() { unsigned char *qshs(); unsigned long r[2]; unsigned char *hash; r[0]=raw_truerand(); r[1]=raw_truerand(); hash = qshs(r, sizeof(r)); return ((int) (*hash)) & 0xff; } </pre>

Fig. 1. UNIX code of the CryptoLib and librand generators.

A similar approach, also for UNIX systems, is followed by the true random generator of librand (see also Fig. 1). Each random byte results from hashing, with SHA, two 32-bit values. Each of these two values results from hashing, also with SHA, twelve other 32-bit values. Each of these twelve values is computed by adding random values until it gets equal or greater than 512. The random values are obtained with a counter that is incremented until a SIGALRM is handled by the current process, and that signal is programmed to be raised 16 665 μ s after the start of counting.

The main drawback of these algorithms is that they are slow and their efficiency is almost independent of the operating system and hardware platform where they run. On the contrary, the algorithm that we will present is very fast, its efficiency improves with faster hardware platforms, and solves the same problem as these others, i.e., it produces high quality random byte sequences without relying on user-provided input or special hardware. We will also show that we don't need to rely on post-processing "whitening" functions, like SHA, to assure the quality of the generator's output.

2.2. Linux kernel-level random generator

Linux systems include a kernel-level random generator that can be accessed by applications as a character device (`/dev/random` or `/dev/urandom`). The kernel gathers environmental noise from the computer's environment, like inter-keyboard timings, inter-interrupt timings, and other events which are both (a) non-deterministic and (b) hard for an outside observer to measure. Randomness from these sources is added to an entropy pool, which is mixed using a CRC-like function. As random bytes are mixed into the entropy pool, the kernel keeps an estimate of how many bits of randomness have been stored into the random number generator's internal state.

When random bytes are desired, they are obtained by taking the SHA hash of the contents of the entropy pool. The SHA hash avoids exposing the internal state of the entropy pool, because it is computationally infeasible to derive any useful information about the input of SHA from its output. The routine decreases its internal estimate of how many bits of true randomness are contained in the entropy pool as it outputs random numbers. If this estimate goes to zero, the routine can still generate random numbers, although less random.

The generator can be used differently with each of the character device interfaces. The `/dev/random` is suitable for use when very high quality randomness is desired, as it will only return a maximum of the number of bits of randomness (as estimated by the random number generator) contained in the entropy pool. The `/dev/urandom` device does not have this limit, and will return as many bytes as are requested. As more and more random bytes are requested without giving time for the entropy pool to recharge, this will result in random numbers that are merely cryptographically strong.

The main advantage of this approach is that it is simple to use, both by user-level applications or kernel modules. Its main drawback, in terms of quality, is that it is very slow to provide really random data, because it must wait for environmental noise. In terms of security its main drawback is that it is trivial, for a system administrator, to deceive applications using the character devices that provide the interface to the generator. In fact, by providing a false device, the administrator has complete control over the random source of many applications without having to take any control of their execution. That is not the case for the previously mentioned generators and for our generator.

A similar approach, but using a per-application randomness pool and a parallel process/thread to gather random data from the system's status information (or random data from the Linux random generator), is described in [13] and implemented in the `cryptlib` library [12].

3. The algorithm

In this section we will present our algorithm for producing high quality random byte sequences. The algorithm is software based, i.e., it does not use any hardware directly, but it uses some uncertainty of the operating system behavior, which can be a consequence of hardware events, as a randomizing factor.

3.1. Overview

Each bit is computed from the least significant bits of a high-precision system timer. Since consecutive bits obtained this way may not be random enough [25], the extraction of consecutive bits is separated by real-time gaps, T_{gap} , several orders of magnitude higher than the real-time clock resolution. Figure 2 presents the generic source code of the random byte generator.

This basic algorithm must be carefully implemented since some systems provide phony or constant values for the least significant bits of the current time [8]. This is not the case for the Pentium's Time-Stamp Register Counter, which is automatically updated by the CPU and can take any 64-bit value.

3.2. Implementation

The implementation of the functions `gap()` and `timeBit()` is not the same for all multi-programmed systems. We will present our implementation for UNIX/Linux systems, using process creation cycles, and the implementation for Solaris/Linux systems, using thread creation cycles. Similar thread creation cycles can also be used in Windows systems.

```
unsigned char randTimeByte ()
{
    int i;
    unsigned char byte;

    for (byte = 0, i = 0; i < 8; i++) {
        gap ();
        byte |= timeBit () << i;
    }
    return byte;
}
```

Fig. 2. Generic source code of the random byte generator.

Both implementations use the same technique for getting accurate real-time values – the current value of the CPU's Time-Stamp Counter. Then we get a random bit from that time by XORing its lower 10 bits (see Fig. 3). We combined more than one bit to increase the randomness of the sample obtained. The number of 10 was chosen taking into consideration (i) the number of clock cycles expended by our implementation of function `gap()` and (ii) the systems where we ran the generator. In all systems that function took more than 20000 clock cycles, then 10 bits represents a rather variable quantity that is less than 5% of the clock cycles expended by `gap()`. We could choose other ways of mixing the bits gathered from the Time-Stamp Counter, but we believe this algorithm is simple and effective, as we will show in Section 5.

The number of bits used, 10, is not a magic value and probably not the only one that could be used successfully. We chose it based on the considerations above stated and the results were very good, as we will see in Section 5. We tested also with only one bit, the least significant bit of the Time-Stamp Counter, but the results were not random enough. Using all the Time-Stamp Counter bits would not be useful because several high order bits do not change in consecutive samples. So, 10 is a value in a range where we should get random samples and no constant bits. But other nearby values, like 9 or 11, could probably be used as well.

To reduce the correlation between consecutive bits obtained with `timeBit()` we introduce a non-null, highly variable T_{gap} . The value of T_{gap} controls the quality and efficiency of the generator: it must be variable and high enough to increase its quality, in terms of randomness of the output bit sequence, and as low as possible to increase performance.

In UNIX/Linux and Windows systems we relied on the uncertainty of the time expended by process/thread creation and termination cycles to introduce real-time gaps in the execution of the generator. The exact combination of system calls used to create real-time gaps is illustrated by the functions `gapPr()` and `gapTh()` of Fig. 4. The function `gapPr()` is for UNIX/Linux systems and uses processes; the function `gapTh()` is for Solaris/Linux systems and uses threads. Hereafter, the name `gap()`

```

unsigned char timeBit ()
{
    unsigned long x;

    asm RDTSC
    asm MOV EAX, x

    return (unsigned char) (1 &
        ((x >> 9) ^ (x >> 8) ^ (x >> 7) ^ (x >> 6) ^ (x >> 5) ^
        (x >> 4) ^ (x >> 3) ^ (x >> 2) ^ (x >> 1) ^ x));
}

```

Fig. 3. Source code, in C and pseudo-assembly, of the function that gets a bit from the Pentium Time-Stamp Counter.

<pre> void gapPr () { int pid, status; switch (pid = vfork()) { case -1: return; case 0: _exit (0); default: waitpid (pid, &status, 0); } } </pre>	<pre> static void rndNullFunc () { thr_exit ((void *) NULL); } void gapTh () { int tid; void * status; thr_create (0, 0, rndNullFunc, 0, THR_BOUND, &tid); thr_join (tid, 0, &status); } </pre>
---	---

Fig. 4. Source code of the functions `gapPr()` (for UNIX/Linux and using processes) and `gapTh()` (for Solaris/Linux and using threads).

will refer to any of these two functions. A T_{gap} delay is the number of clock cycles that a `gap()` function takes to complete from the generator's point of view.

The function `gapPr()` creates a new process, using the lightweight system call `vfork()`, and the original process waits for the termination of the new one, which terminates immediately. The function `gapTh()` acts similarly to `gapPr()` but uses threads instead of processes.

The creation and termination of processes or threads are complex system operations, that may follow different execution paths, and the time they take to execute depends on many factors. Some of these factors are related with the execution of the operating system, such as the allocation and duplication of per-process resources and process dispatching, and others are related with the CPU execution, such as cache misses, interrupt handling, context switching, etc. See the next section for a more detailed analysis of the some factors influencing the time `gapPr()` takes to execute.

Note that each instruction on a Pentium processor takes a few clock cycles to execute. Thus, even a small variation in the total number of instructions executed during the call to `gap()`, but not necessarily related with it, may yield a significant variation in T_{gap} delays produced by `gap()`.

4. Entropy analysis

In this section we analyze the entropy of T_{gap} delays produced by the implementations of function `gap()` described in the previous section. First we present the rationale for using process or thread creation and termination cycles. Next we analyze real sequences of T_{gap} delays produced in several systems in order to assess their variation and unpredictability on a secure environment, i.e., without considering attacks. Finally, we address the security of our generator against attacks attempting either to clone, monitor, reproduce or control its activity.

4.1. Rationale

Our hypothesis is that, by using `gapPr()` or `gapTh()`, we should get many different values for T_{gap} , and that given past T_{gap} values, it should not be possible

to predict future ones. It is difficult to formally demonstrate this hypothesis without analyzing the code of operating system kernels and instrumenting them to get accurate execution profiles on particular hardware platforms and workload conditions. Nevertheless, there are some aspects in the design of kernels, namely UNIX kernels [2,17], and experimental results gathered from UNIX kernels' profiling [19,27], that account for the validity of our hypothesis. These are the following.

The child process created by `gapPr()` gets a new, unused identifier (PID). There are different ways of allocating PIDs: incrementing the last allocated PID and checking if it is not being used, looping until finding a free one [2], or getting a PID from a pre-arranged pool of free PIDs, which must be rebuilt when exhausted [17]. In any case, the time to allocate a new PID is not always the same.

The code of the kernel running on behalf of a process executes in different execution levels, or processor priority levels, to prevent the occurrence of interrupts during critical activity [2,17]. An accurate profiling of the 386BSD kernel [19] showed that the functions that set the process priority level, `spl*` functions, yield a slight variation in their execution time. This variation is of some microseconds on a Intel architecture with a 40 MHz clock. The analysis of the source code of the FreeBSD 2.1.6 and BSD4.3 kernels, and according to the profiling results presented by McRae in [19], showed that `spl*` functions are called a great deal of times on each `vfork/_exit/waipid` cycle. Assuming that a similar behavior should be expected in all UNIX kernels, and that the variation in the execution time of functions setting priority levels can also occur in other hardware platforms, T_{gap} delays produced by this implementation of `gapPr()` should yield a slight but unpredictable variation due to the modification of processor priority levels.

Performance evaluations on a DECstation 5000/200 running Ultrix 4.2 [27] showed that the handling of real-time ticks preempts the measurement process, executing on an otherwise idle machine, during a highly variable time. Preemption times span from 10 μ s up to 500 milliseconds, 65% of them last between 20 μ s and 25 μ s, and only 5% of them take more than 100 μ s. Extrapolating these results to other UNIX systems and hardware platforms, one may expect to get some T_{gap} delays unpredictably increased when interrupted by the handling of real-time ticks. The percentage of T_{gap} delays affected this way depends on the system where `gapPr()` or `gapTh()` are executed, and in particular on the frequency of tick interrupts. It also depends on the workload of the system, which affects the time expended by the tick handler or by kernel housekeeping processes wakened by the tick handler, like the scheduler or the swapper.

The new process created by `gapPr()` gets its own page map, where its parent's PTEs are copied into. Similarly, the new thread created by `gapTh()` gets its own stack address space, for which some page map space must be allocated. Page maps are variable-size memory areas that are allocated in limited memory space defined by the kernel. Different kernels may use different allocation algorithms; for instance, the BSD kernel uses a first-fit algorithm [17]. In any case, the free space for allocating page tables depends on the running processes, and in particular on the way they grow

or shrink their address spaces. Hence, T_{gap} delays depend on the time expended by the algorithm that allocates page maps, and on the current address space used by all processes.

Note that we explicitly chose process/thread creation for introducing T_{gap} delays because they are heavyweight time consuming kernel actions. Simpler kernel actions could be too insensible to hardware events, could interact less with interrupt masking, and could have fewer different execution paths, thus potentially yielding less random T_{gap} delays.

4.2. Analysis of real T_{gap} sequences

Previously we presented the rationale for using process or thread creation and termination cycles to produce variable and unpredictable real-time gaps in the execution of our generator. In this section we measure such gaps in four systems, presented in Table 1, to verify if, in those systems, they are in fact variable and unpredictable.

For evaluating the evolution of T_{gap} along many consecutive invocations of `gapPr()` and `gapTh()` we produced 10 sample sequences of 10 000 T_{gap} values on each of the tested systems, lightly loaded. Then we chose, for each system, the worst possible sample, i.e., the one with the smallest standard deviation (σ).

Table 2 presents the minimum, average, maximum and σ values obtained for the T_{gap} sequences. The probability of each of value in the sequences is presented in the charts of Fig. 5. Figure 6 shows a full histogram of the sequences and highlights particular areas showing a maximum of 500 T_{gap} values. To give a notion of the existing correlation between consecutive T_{gap} values, Table 3 presents the number of constant runs of T_{gap} values, and Table 4 presents the number of equal runs of T_{gap} values.

The values of Table 2 show that T_{gap} delays are not constant, as expected, and take many thousands of clock cycles. From the charts of Fig. 5 we see that T_{gap} delays take a great many values, even in a lightly loaded scenario; no value was observed in more than 1% of all observations. Finally the charts of Fig. 6 clearly show that consecutive values obtained for T_{gap} do not follow a clear pattern, thus not allowing one to predict them with precision given the previous one.

The results presented in Table 3 show that consecutive T_{gap} delays are rarely equal, and that their occasional constancy does not last long. This is very important for the

Table 1
Systems where we implemented and tested the random byte generator

	Operating system	CPU		Cache L2	Cache L1	RAM
				(kbytes)	data/code (kbytes)	(Mbytes)
S_1	Linux 7.1 (2.4.2-2smp)	P II	233 MHz	512	16/16	128
S_2	Linux 7.1 (2.4.2-2smp)	P III	1 GHz	256	16/16	256
S_3	Windows XP Prof.	P II	331 MHz	512	16/16	128
S_4	Windows 2000 Prof.	P III	1 GHz	256	16/16	256

quality of the generator, as it shows that local non-randomness scenarios are highly unlikely to occur.

The results presented in Table 4 show that there is a non null, although reduced, probability of getting short, equal runs of T_{gap} delays, mainly in Linux systems and in system S_2 . This means that there may be some short-time local non-randomness scenarios, but, from our observation, such equal runs do not occur in a predictable way, so they cannot be used to predict the output of the generator; they may only slightly reduce the statistical quality of its output.

Note that the very small, though non-null, correlation between consecutive T_{gap} delays, which potentiates the local non-randomness of the generator, is not critical for its security. Such correlation is as unpredictable as T_{gap} delays are, and even other applications running “in parallel” with our generator cannot be sure, when

Table 2

Examples of the real-time T_{gap} , in clock cycles, expended by functions `gapPr()` and `gapTh()` in 10000 consecutive calls in the test systems

System	gap () function	T_{gap} (μs)			
		Minimum	Average	Maximum	σ
S_1	gapPr ()	30 242	31 458	418 317	5629
S_2		22 242	23 243	1 984 484	20 600
S_3	gapTh ()	114 806	132 611	867 380	24 119
S_4		112 274	120 440	501 447	16 156

Table 3

Number of constant runs of length 2 and 3 of T_{gap} produced by functions `gapPr()` and `gapTh()` in 10000 consecutive calls in the test systems

System	gap () function	Constant runs of length	
		2	3
S_1	gapPr ()	41	0
S_2		20	0
S_3	gapTh ()	5	0
S_4		2	0

Table 4

Number of runs of length 2 of T_{gap} , produced by functions `gapPr()` and `gapTh()`, occurring 2, 3, 4 and 5 times in 10000 consecutive calls in the test systems

System	gap () function	Runs of length 2 occurring					
		2 times	3 times	4 times	5 times	6 times	7 times
S_1	gapPr ()	458	19	0	0	0	0
S_2		761	163	49	24	16	0
S_3	gapTh ()	5	0	0	0	0	0
S_4		1	0	0	0	0	0

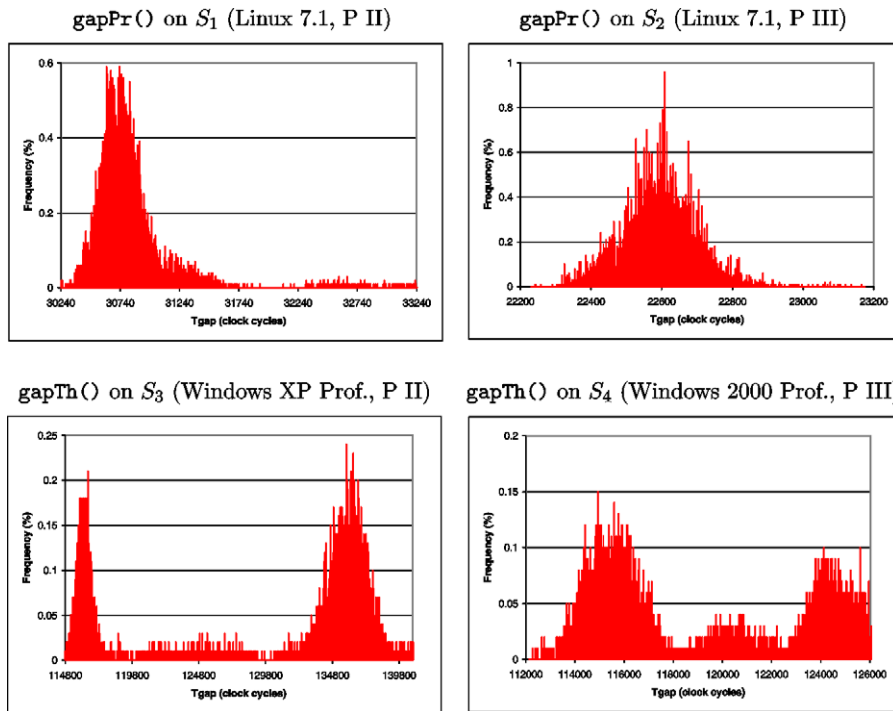


Fig. 5. Charts of the observed frequency of T_{gap} , in clock cycles, expended by functions `gapPr()` and `gapTh()` in 10 000 consecutive calls in the test systems. Each chart shows only 95% of the lower values observed for T_{gap} in order to zoom in the areas where most values occur; the absent 5% values are a long way off to the right. Note also that the charts use different scales to better illustrate the observed probabilities.

producing T_{gap} delays, that they are equal to the ones produced “simultaneously” by our generator.²

Concluding, in the test systems the functions `gapPr()` or `gapTh()` can be used to reduce the correlation between consecutive time bits obtained by `timeBit()`. Note that we reach this conclusion after analyzing sequences of T_{gap} delays produced in a lightly loaded execution scenarios, thus adverse test environments. Increasing the number of processes waiting for the processor or incrementing the number of hardware interruptions (with user input or network interaction) should further increase the entropy of T_{gap} delays.

²In single-processor systems, where there is a pseudo-parallelism between running processes, the quoted expressions should be interpreted as “immediately after” or “immediately before”.

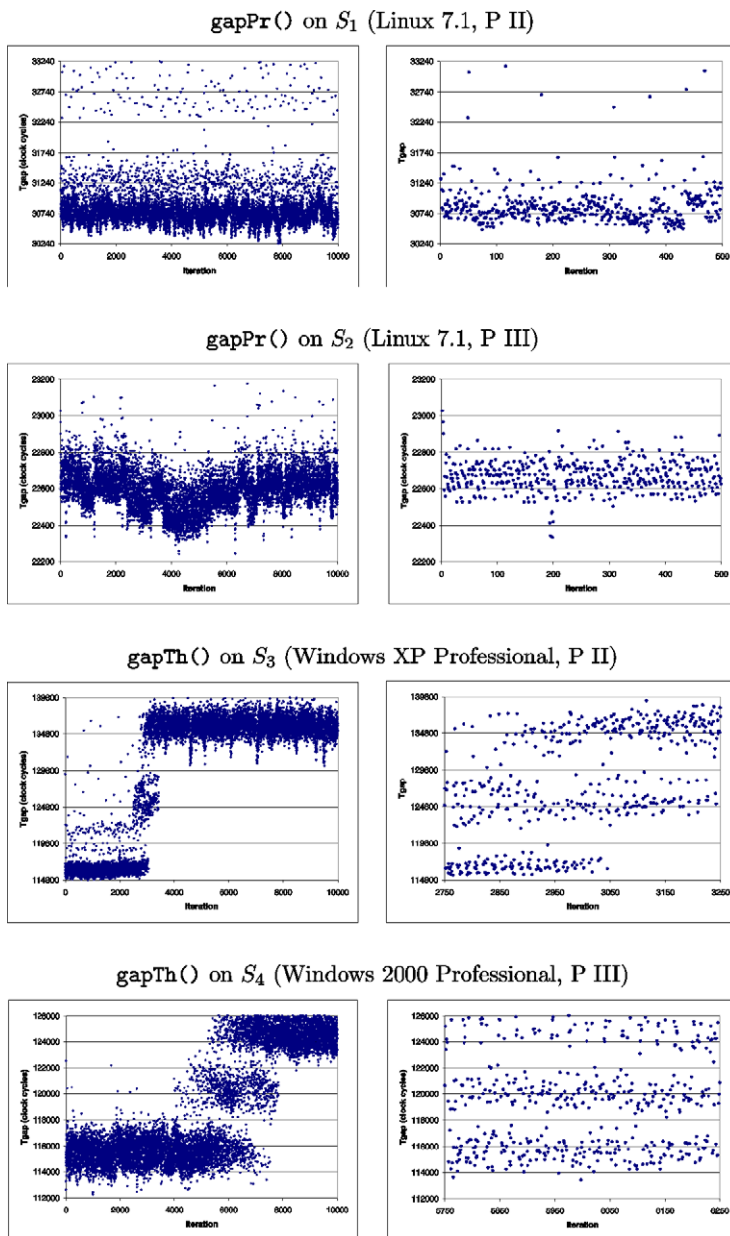


Fig. 6. Histograms of observed T_{gap} expended by functions $gapPr()$ and $gapTh()$ in the full 10000 consecutive iterations in the test systems, and a highlight showing a maximum of 500 of those iterations. The scale for T_{gap} covers 95% of the lower values observed. Note also that the charts use different scales to better illustrate the observed values.

4.3. Security analysis

True random generators are very valuable for producing secret data for cryptographic applications. But for these applications it is not enough to get pure random data, we wish also that the attackers could not derive it by any means. In this section we will analyze the security of our generator to access its strength against attacks trying to guess the values it produces (or produced sometime in the past), or attacks trying to influence its behavior in order to facilitate guessing attacks. Such analysis involves the following aspects: cloning, reproduction, monitoring, and control of the activity of generator.

4.3.1. Cloning

An operating system is a complex state machine, which is partially hidden from user applications. Its state evolves in response to hardware events, like timer ticks and I/O interrupts, or user application's requests. We believe, although we cannot prove, that it is impossible to have, in two machines with exactly the same hardware, the same operating system state and the same running applications at the same time. For instance, a small difference in the frequency of the systems' hardware clock is enough, after some time, to make the systems to react in a different clock cycle to an event occurring in both systems in the same real instant (something that is, by itself, already hard to get). Therefore, we believe it is impossible to guess sequences produced by our generator by cloning the systems where they are produced.

We did a very simple experience to evaluate the difficulty of getting twice the same bit sequence in the same computer. We set up the system to run the bit generator just after booting a stand-alone computer in multi-user mode and we generated several sequences this way after hardware resets. In all cases the boot sequences ran without user intervention. The result was a set of uncorrelated bit sequences, showing how difficult it is to reproduce the same behavior in separate runs of the bit generator in very similar execution scenarios. We think this simple experience shows how hard it should be to fully clone a computer system, particularly if connected to a network.

4.3.2. Monitoring and reproduction

In multi-user workstations users may not trust each other, thus it is important to show that an attacker cannot monitor the activity of our generator in order to guess the sequences it produces. Naturally, we are not concerned with attacks performed by administrators, as these can fully access all system resources. Thus, they do not need to use sophisticated techniques to guess sequences produced by our generator, since they can get the random sequences it produces, or other secret data, directly from the address space of processes, even when using some sort of random relocation of sensible data (like the randomness pool in `cryptlib` [13]). On the contrary, we are concerned with attacks performed by normal users, eventually using confidential data like audit trails.

By monitoring the activity on the system, and in particular the existing processes, it is easy to detect when our generator is working, unlike for the generators of

CryptoLib and librand. However, the observation of processes' creation and death times does not help the attacker, as the bits gathered by our generator depend on the instants when it uses the processor and calls the RDTSC instruction.

Audit trails maintain a history of processes' requests that may be critical for the security of the system, including stored data. If the result of a RDTSC could be recorded in an audit trail, then the security of our generator could be compromised if an attacker could access audit records. However, the current value of the Time-Stamp Counter is not confidential information and the RDTSC instruction is not privileged. Thus, an operating system cannot detect a call to RDTSC in order to add a related entry to an audit trail.

4.3.3. Controlling the Time-Stamp Counter

Another important aspect concerning the security of our generator is whether or not an attacker is able to control the time bits gathered by our generator by means of a manipulation of Time-Stamp Counter register. However, that is not possible, because the register can only be read (with RDTSC), and is written only by the processor. The only way to deterministically change the value of the counter is by resetting the processor, setting the counter to zero.

5. Statistical analysis

In this section we evaluate the quality of the bit sequences produced by our generator, and the generators of CryptoLib and librand, using a set of statistical tests. These tests evaluate how close bit sequences produced by a particular generator are from the ones expected to be generated by a true random bit generator.

As statistical tests we used most of the tests described in [3] and included in the CRYPT-X package [11]: the frequency test, the serial test, the poker test, the binary derivative test, the runs test, and the universal test [4,6,10,18,21]. The latter test uses the normal distribution, while the others use the χ^2 distribution. Briefly, the frequency test evaluates the proportion of 0's and 1's in a bit stream; the serial test evaluates the proportion of all possible transitions between consecutive bits in a bit stream; the poker test evaluates the proportion of all different non-overlapping m -bit hands in a bit stream; the binary derivative test evaluates the proportion between the 4 possible overlapping 2-bit tuples in a bit stream; the runs test evaluates the proportion of blocks (runs of 1's) and gaps (runs of 0's) of several lengths in a bit stream; finally, the universal test evaluates the entropy of a stream of bit blocks with length L .

The tests that use the χ^2 distribution compute a χ^2 value that should be lower than χ^2_α , where $\alpha \in [0, 1]$ is the significance level of the test. The universal test analyzes long sequences of bit blocks and computes the value of a function (f_{TU}) giving the quality of the sequence generator for a given significance level α (rejection rate ρ in [18]). We evaluated the randomness quality of byte sequences by computing the

highest significance level (α_{\max}) allowing the statistical tests to consider the generator a good one. The usual or recommended values for α are between 0.001 and 0.05 [3,11] for χ^2 tests, and for the universal test it is recommended to use a value between 0.001 and 0.01 [18].

We produced one byte sequence with a total length of 300 000 bytes per system and per generator. All byte sequences were produced with low system entropy and stored in virtual memory while being produced (to reduce I/O activity during its generation). For analyzing such sequences we used 8-bit hands (bytes) in the poker test, set the maximum run length of the runs test to 16 ($k = 16$), and used 6-, 7- and 8-bit blocks in the universal test³ ($L \in \{6, 7, 8\}$), with the initialization steps $Q_{L=6} = 640$, $Q_{L=7} = 1280$ and $Q_{L=8} = 2560$).

5.1. Statistical evaluation

As mentioned in [18], real random byte sequences cannot be compressed or, conversely, if it is possible to compress a given byte sequence, than it is not random. We tried to compress the sequences produced by our generator with two popular UNIX compressing tools – `compress` and `gzip` – but we did not succeed. However, these tools are not designed to compress their inputs at all cost, but instead to trade off their execution-time against the compression they can economically achieve. As a consequence, this compression test does not allow us to fully assess the randomness of the sequences, but it shows, at least, that they do not contain obvious patterns.

Table 5 shows the α_{\max} values derived from the statistical tests of the byte sequences produced by our generator in the four test systems. The table shows also the same values for the other two generators, that were only tested in two systems, S_1 and S_3 , since their algorithm basically depends on the operating system, and not on the processor speed. For simplicity, we do not present in the table the values χ^2 and f_{TV} computed by the statistical tests.

The results of the statistical tests presented in Table 5 show that all sequences can be considered as good random sequences, since most of the values of α_{\max} are higher than the usual or recommended values. The rejection exceptions occur only in three cases: one with the universal test for S_4 and $L = 6$; and the other two with the poker and the runs tests for S_2 . The good results observed with the universal test are particularly relevant, since this test is able to detect any one of a very general class of possible defects (deviations from the statistics of a binary symmetric source) the generator may have, including all the ones that are detected by the other tests we used.

The result obtained for our generator, regarding the statistical tests, could be improved by mixing the bits obtained from the Time-Stamp Counter using a more sophisticated algorithm, e.g., using an hashing function like SHA, the complex algorithm used by the kernel generator of Linux, or even the mixing function used by

³The author recommends to choose the parameter L between 6 and 16.

Table 5

Statistical test of byte sequences generated by our generator, using the functions `gapPr()` or `gapTh()` in the four test systems, and the generators of `CryptoLib` and `librand` in two of those systems. The values refer to sequences with 300 000 bytes

System	Algorithm	α_{\max} for statistical tests							
		Frequency	Serial	Poker	Binary	Runs	Universal		
				$m = 8$	derivative	$k = 16$	$L = 6$	$L = 7$	$L = 8$
S_1	RDTSC & <code>gapPr()</code>	0.325	0.485	0.114	0.835	0.423	0.798	0.548	0.828
S_2		0.544	0.815	≈ 0	0.776	≈ 0	0.967	0.017	0.878
S_3	RDTSC & <code>gapTh()</code>	0.514	0.517	0.260	0.259	0.918	0.722	0.999	0.206
S_4		0.380	0.248	0.135	0.701	0.027	≈ 0	0.980	0.790
S_1	<code>CryptoLib</code>	0.011	0.036	0.151	0.035	0.265	0.533	0.108	0.248
	<code>librand</code>	0.386	0.665	0.264	0.660	0.074	0.316	0.009	0.298
S_3	<code>CryptoLib</code>	0.591	0.138	0.639	0.083	0.841	0.602	0.867	0.245
	<code>librand</code>	0.193	0.428	0.019	0.323	0.740	0.564	0.883	0.219

`cryptlib` [13]. But, by not doing so, we could better evaluate the quality of our source of randomness.

Comparing our generator with the other two we can see that none of them is always better than the others regarding all statistical tests, and they all have good statistical quality. This is a very important fact, since our generator does not use a cryptographic strong hash function for whitening input bits or randomness pools in order to get a “more random” output, like the other generators do (using SHA). This means that, at least in the systems that we used and with the tests that we ran, the source of randomness used by our generator is good enough, by its own, to produce high quality random sequences, not requiring the help of cryptographic strong mixing functions (that, by design, are suitable to produce sequences with good statistical quality).

6. Performance evaluation

Table 6 shows the performance of our generator, as well as the performance of the generators of `CryptoLib` and `librand`, evaluated on lightly loaded machines. To evaluate the performance we run the generators 10 times, producing 10 000 bytes each time, for reaching an average speed per byte.

In theory, the performance of `CryptoLib` and `librand` is bounded by their algorithms, independently of the system where they run. The maximum theoretical performance of these generators, assuming a null execution time for the SHA function, should be:

$$\text{CryptoLib: } performance_{\text{maximum}} = \frac{1000}{11 \times 16.665} \approx 5.5 \text{ bytes/s,}$$

$$\text{librand: } performance_{\text{maximum}} = \frac{1000}{24 \times 16.665} \approx 2.5 \text{ bytes/s.}$$

However, from the values of Table 6 we see that the real performance of these algorithms is higher than the maximum theoretical, but rather constant independently of the speed of the machine's processor. The reason for the apparent mismatch between the maximum theoretical values and the real ones is the fact that both operating systems do not handle with great precision the timers these algorithms use.

On the contrary, the performance of our generator depends on the system where it runs: it is more efficient in systems where it is faster to execute process/thread creation and termination cycles. Comparing the performance of our generator with the performance of the other two generators we conclude from the values of Table 6 that, in the machines we used, our generator is about 40 to 300 times faster than `CryptoLib`, and about 30 to 340 times faster than `librand`.

Another important issue regarding performance is how the generators affect the overall performance of a heavily loaded system. Our generator runs mostly in kernel mode, whereas the generators of `CryptoLib` and `librand` run mostly in user mode. If most of the system load is due to kernel activity, for instance, due to intensive I/O, then the execution of our generator should be more noticeable than the execution of the other two generators, as it may delay external I/O requests. If, on the contrary, most of the load is due to intensive CPU usage, processes are more likely to execute during their full time quanta. Consequently, the generators of `CryptoLib` and `librand` should be more noticeable than ours, since they would tend to burn a lot of time quanta incrementing counters until getting a alarm signal, whereas our generator takes less time quanta for getting similar results.

Table 6

Performance of our improved generator, using `RDTSC` and either `gapPr()` or `gapTh()`, and the generators of `CryptoLib` and `librand`

System	Algorithm	Performance (bytes/s)
S_1	<code>CryptoLib</code>	18.1
	<code>librand</code>	16.7
	<code>RDTSC & gapPr()</code>	1037.4
S_2	<code>CryptoLib</code>	18.2
	<code>librand</code>	16.2
	<code>RDTSC & gapPr()</code>	5480.2
S_3	<code>CryptoLib</code>	7.3
	<code>librand</code>	9.3
	<code>RDTSC & gapTh()</code>	312.4
S_3	<code>CryptoLib</code>	12.1
	<code>librand</code>	12.1
	<code>RDTSC & gapTh()</code>	992.0

7. Using the generator in other systems

An important question about our generator is whether or not it may be used in any other multi-programmed system without any problems. However, our generator depends on many aspects of the system where it runs. Thus, to answer this question one must repeat the design and evaluation process used in the previous section, which can be summarized as follows:

1. Design a proper implementation of the generator for the target system. In such design one must first evaluate the *effective* real-time clock resolution available for applications. If the real-time clock resolution is much higher than the time expended by each machine instruction, then our algorithm must be modified.

After that, one should choose complex system operations, like process creation and termination cycles, taking a variable execution time higher than the real-time clock resolution, and use them to implement the function $\text{gap}()$.

2. Analyze the range of T_{gap} delays produced by the chosen implementation of $\text{gap}()$ in low or normal workload scenarios. The value of T_{gap} controls the efficiency and quality of the generator: it must be high enough to increase its quality, in terms of randomness of the generated bit sequence, and as low as possible to increase performance.

The implementation of $\text{gap}()$ should also produce many T_{gap} delays, the more the better. The distribution of such delays is also important for the quality of the generator, which should be improved by uniform distributions of T_{gap} delays in a limited range. The correlation between the delays should also be analyzed.

3. Analyze the byte sequence produced by the generator using statistic tests, like the ones we used in Section 5.

8. Conclusions

We presented a practical and efficient software algorithm for producing true random byte sequences in multi-programmed systems. We implemented the algorithm in UNIX/Linux and Windows systems but it may also be implemented in other multi-programmed systems. The algorithm is software based, i.e., it does not use any hardware techniques directly, but it uses some uncertainty of the operating system behavior, which can be a consequence of hardware events, as a randomizing factor.

Each random bit is extracted from a system time counter, namely from a register of the Pentium processor (Time-Stamp Counter), and the extraction of consecutive random bits is separated by random real-time gaps higher than the real-time clock resolution. In particular for the systems we tested, we used the uncertainty of the process/thread management as the source of random real-time gaps.

Since the algorithm depends on the UNIX/Windows process/thread management, we ran all the tests in an adverse scenario, i.e., with light load to minimize the entropy regarding process management. Increasing the number of processes waiting for the processor and increasing the number of interrupts (due to user input or network interaction) should increase the randomness of the real-time gaps and therefore, the randomness of the bits generated by our algorithm.

We evaluated the quality of our generator with several statistical tests. These tests proved that, at least in the tested systems, our generator produces good random byte sequences, being similar in quality to the sequences produced by `CryptoLib` and `librand`, but without needing to use any cryptographic whitening function, like the others, to output unpredictable and statistically good random data. Concerning performance, the speed of our generator increases with the speed of the processor, while the speed of `CryptoLib` and `librand` is rather constant for each operating system, independently of the processor where it runs.

References

- [1] G.B. Agnew, Random sources for cryptographic systems, in: *Advances in Cryptology – EUROCRYPT'87 Proceedings*, Springer, Berlin, 1987, pp. 77–81.
- [2] M.J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.
- [3] H. Beker and F. Piper, *Cipher Systems: The Protection of Communications*, Northwood Books, London, 1982.
- [4] J. Carroll and L. Robins, Computer Cryptanalysis, Technical Report 223, Dept. of Computer Science, The University of Western Ontario, London, Ontario, Canada, 1988.
- [5] D. Davis, R. Ihaka and P. Fenstermacher, Cryptographic randomness from air turbulence in disk drives, in: *Advances in Cryptology – CRYPTO'94 Proceedings (LNCS 839)*, Springer, Berlin, 1984.
- [6] E.P. Dawson, Design and cryptanalysis of symmetric ciphers, PhD thesis, Queensland University of Technology, Australia, 1991.
- [7] T. Dierks and C. Allen, The TLS Protocol, Version 1.0. RFC 2246, January 1999.
- [8] D. Eastlake, S. Crocker and J. Schiller, Randomness Recommendations for Security, RFC 1750, December 1994.
- [9] I. Goldberg and D. Wagner, Randomness and the netscape browser, *Dr. Dobb's Journal* (January) (1996).
- [10] I.J. Good, On the serial test for random sequences, *Ann. Math. Statist.* **28** (1957), 262–264.
- [11] H. Gustafson, E. Dawson, L. Nielsen and W. Caelli, A computer package for measuring the strength of encryption algorithms, *Computers & Security* **13**(8) (1994), 687–697.
- [12] P. Gutmann, cryptlib Free Encryption Library, <http://www.cs.auckland.ac.nz/~pgut001/cryptlib>.
- [13] P. Gutmann, Software generation of practically strong random numbers, in: *Proc. of the 7th USENIX Security Symp.*, 1998.
- [14] J.M. Hammersley and D.C. Handscomb, *Monte Carlo Methods*, Monographs on Applied Probability and Statistics, Chapman and Hall, 1964, ISBN 0 412 15870 1.
- [15] D. Harkins and D. Carrel, The Internet Key Exchange (IKE), RFC 2409, November 1998.
- [16] J.B. Lacy, D.P. Mitchell and W.M. Schell, Cryptolib: cryptography in software, in: *Proc. of the 4th UNIX Security Symposium*, USENIX Association, 1993, pp. 1–17.

- [17] S.J. Leffler, M.K. McKusick, M.J. Karels and J.S. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley, 1989.
- [18] U.M. Maurer, A universal statistical test for random bit generators, *Journal of Cryptology* **5**(2) (1992), 89–105.
- [19] A. McRae, Hardware profiling of kernels, in: *Proc. of the USENIX Winter Conf.*, San Diego, CA, USA, 1993, pp. 375–386.
- [20] A.J. Menezes, P.C. van Oorschot and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.
- [21] A.M. Mood, The distribution theory of runs, *Ann. Math. Statist.* **11** (1940), 367–392.
- [22] The International PGP Home Page, <http://www.pgpi.org>.
- [23] C. Plumb, Truly Random Numbers, *Dr. Dobb's Journal* (November) (1994).
- [24] T. Ritter, The efficient generation of cryptographic confusion sequences, *Cryptologia* **15**(2) (1991), 81–139.
- [25] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*, second edition, Wiley, 1996.
- [26] Secure Hash Standard, NIST FIPS PUB 180, Washington, DC, USA, April 1993.
- [27] M. Shand, Measuring Unix kernel performance, in: *AUUG'91 – Australian Unix User's Group Meeting*, 1991.
- [28] Secure Shell (secsh) Internet Drafts, <http://www.ietf.org/ids.by.wg/secsh.html>.
- [29] J.G. Steiner, C. Neuman and J.I. Schiller, Kerberos: an authentication service for open network systems, in: *Proc. of the USENIX Winter Conf.*, Dallas, TX, USA, 1988, pp. 191–202.