Syndrome-Fortuna: A viable approach for Linux random number generation

Sérgio Vale Aguiar Campos[1], **Jeroen van de Graaf**[1], **Daniel Rezende Silveira**[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte (MG) – Brazil

Abstract. *This work presents a random number generator based on the intractability of an NP-Complete problem from the area of error-correcting codes. It uses a non-heuristic approach for entropy collection, taken from the Fortuna design philosophy. We implemented the new generator inside the Linux kernel, providing an alternative system interface for secure random number generation.*

1. Introduction

Random number generators are the basis of several cryptographic systems. Its output must be unpredictable by potential adversaries and should be produced fast and efficiently. The most common way to achieve this is by using algorithms to mix and expand the outcome of entropy sources. These algorithms are called pseudo-random number generators (PRNGs). The quality of these generators and their applicability to protocols and security applications have been widely studied in recent years.

In this work we present a PRNG based on the Fortuna architecture, proposed by [Ferguson and Schneier, 2003]. Fortuna presents a new scheme for system events collection, that does not use any heuristics, avoiding entropy estimation problems. Its mixing function is the AES algorithm, considered strong and efficient.

The PRNG we propose, called Syndrome-Fortuna, changes the mixing function of Fortuna, using the syndrome decoding algorithm proposed by [Fischer and Stern, 1996]. We consider this an improvement, since the syndrome problem is known to be NP-Complete, and has a formal proof of its strength.

We implemented Syndrome-Fortuna inside the Linux kernel version 2.6.30, providing an user interface for random numbers besides /dev/urandom. As we expected, our generator is slower than the original Linux random number generator (LRNG). But it does not use any entropy estimation heuristics and applies a strong and formally proved algorithm as its core function.

1.1. Randomness concept

Random number generators emerged, initialy, for use in simulations and numerical analysis. These applications require efficiency and, especially, *uniformity*. The cryptography evolution brought a new requirement on PRNGs. Secure applications needed secret keys, generated randomly, to allow users to communicate safely. Since then, *unpredictability* has become a fundamental requirement for pseudorandom number generators.

The only way to ensure that a generator seed is non-deterministic is by using real sources of randomness. External sources of randomness collect data from presumably unpredictable events, such as variations in pressure and temperature, ambient noise, or timing of mouse movements and keystrokes. The concept of unpredictability is related to human inability to predict certain complex phenomena, given its chaotic or quantum essence.

Before using the collected randomness, however, it is necessary to *estimate* it. The goal is to prevent that the internal state of the generator is updated with values potentially easy to discover. In 1996 a flaw in the random number generator of *Netscape* browser allowed that keys used in SSL connections were discovered in about one minute. The problem, revealed in the work of [Goldberg and Wagner, 1996], was the use of system time and the *process id* as sources of randomness. Even when the browser used session keys of 128 bits, considered safe, they possessed no more than 47 bits of randomness, very little to prevent that opponents using brute force could find the key value.

Entropy estimation is one critical point of random number generators design, because their security level is directly related to estimates precision. As we shall see, the approach of entropy collection proposed by [Ferguson and Schneier, 2003] and adapted in this work minimizes the problem of possible inaccuracy in the estimation of entropy.

2. Construction of a cryptographically secure random number generator

2.1. One-way functions

Intuitively, a function f is one-way if it is easy to compute but difficult to invert. In other words, given x, the value $f(x)$ can be computed in polynomial time. But any feasible algorithm that receives $f(x)$ can generate an output y such that $f(y) = f(x)$ with only negligible probability.

The existence of one-way functions is not proved. It is known that if $P = NP$ they certainly do not exist. But it is unclear whether they exist if $P \neq NP$. However, there are several functions that are believed to be unidirectional in practice. One of them is the SHA-1 hash function, used inside the Linux random number generator. There are also functions that are conjectured to be unidirectional. Some examples are the subsets sum problem and the discrete logarithm calculation. These functions belong to the class NP, and supposing $P \neq NP$, and are unidirectional under some intractability assumption.

The main difference between the two types of one-way functions, besides the theoretical basis, is the computational cost. Functions based on intractable mathematical problems require, in general, a larger amount of calculations per bit generated. As shown in [Impagliazzo et al., 1989], cryptographically secure pseudorandom number generators exists if, and only if, one-way functions exists.

In the generator presented in this paper we use the algorithm proposed by [Fischer and Stern, 1996] as our one-way function. It is based on the syndrome decoding problem, a NP-complete problem from the area of error-correcting codes.

2.2. Syndrome decoding problem

In coding theory, coding is used to transmit messages through noisy communication channels, which can produce errors. *Decoding* is the process of translating (possibly corrupted) *messages* into *words* belonging to a particular *code*. A *binary linear code* is an error-correcting code that uses the symbols 0 and 1, in which each *word* can be represented as a *linear* combination of others, and each word has a *weight*, ie, a number of 1 bits, defined.

A binary linear code (n, k, d) is a subspace of $\{0, 1\}^n$ with 2^k elements in which every word has weight less than or equal to d. The information rate of the code is k/n and it can correct up to $\lfloor \frac{d-1}{2} \rfloor$ errors. Every code can be defined by a parity matrix of size $n \times (n - k)$ which multiplied (mod 2) by a vector of the code $x = \big(x_1, x_2, \ldots, x_n\big)$ results in an all zero vector $s = \big(0, 0, \ldots, 0\big)$ of size $(n-k)$, called *syndrome*. Conversely, when the word multiplied by the parity matrix does not belong to the code, the value of the syndrome is nonzero.

A randomly filled parity matrix defines a random binary linear code. For this code, there is no efficient algorithm known that can find the closest word to a vector, given a nonzero syndrome. Another difficult problem, known as *syndrome decoding*, is to find a word of certain weight from its syndrome.

The latter problem is NP-Hard and can be described as follows: let a binary matrix $A = \{a_{ij}\}$ of size $n \times (n - k)$, a non-zero syndrome vector s and a positive integer w. Find a binary vector x with weight $|x| \leq w$, such that:

$$\big(x_1, x_2, \ldots, x_n\big) \cdot \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n-k} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n-k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n-k} \end{pmatrix} = \big(s_1, s_2, \ldots, s_{n-k}\big) \pmod 2$$

The case in which we seek the vector x with weight $|x| = w$ is NP-complete [Berlekamp et al., 1978].

The fact that a problem is NP-Complete guarantees that there is no polynomial time algorithm for solving the worst case, under the assumption that $P \neq NP$. Many problems, however, can be solved efficiently in the average case, requiring a more detailed study of each instance of the problem.

2.2.1. Gilbert-Warshamov Bound

To evaluate the hardness of a specific instance of the syndrome decoding problem we will use a concept extensively studied and reviewed by [Chabaud, 1994], under which the most difficult instances of the problem of syndrome decoding for random codes are those with weights in the vicinity of the Gilbert-Warshamov bound of the code.

The Gilbert-Warshamov bound λ of a code (n, k, d) is defined through the relation $1 - k/n = H_2(\lambda)$ where $H_2(x) = -x \log_2 x - (1 - x) \log_2(1 - x)$ is the binary entropy function.

According to the analysis of [Fischer and Stern, 1996], there is, on average, a vector for each syndrome when the weight is around the Gilbert-Warshamov bound of the code. That is, the difficulty of finding a word is a function of the weight increasing until the Gilbert-Warshamov bound, and decreasing thereafter. Thus, it is possible to define a hard instance of the syndrome decoding problem when the weight is near the Gilbert-Warshamov bound.

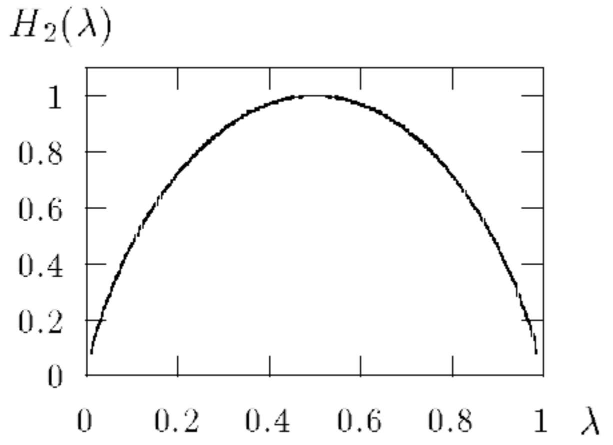

Figure 1. Gilbert-Warshamov bound, defined by the binary entropy function.

2.2.2. Formal definitions

Definition A function $f : \{0,1\}^* \to \{0,1\}^*$ is considered strongly unidirectional if the following conditions apply:

- *Easy to compute*: there is a deterministic polynomial-time algorithm A such that for every input x, an output $A(x) = f(x)$ is computed.
- *Hard to invert*: for all probabilistic polynomial-time algorithm A' and every positive polynomial $p(n)$ large enough,

$$Pr(A'(f(X_n)) \in f^{-1}(f(X_n))) < \frac{1}{p(n)}$$

where x_n is random and uniformly distributed over $\{0,1\}^n$

Let us consider a collection of functions related to the problem of decoding the syndrome.

Definition Let ρ be in $]0,1[$ and δ be in $]0,1/2[$. A collection $SD(\rho,\delta)$ is a set o functions f_n such that:

$$\begin{aligned} D_n &= \{(M,x), M \in \lfloor \rho n \rfloor \times n, x \in \{0,1\}^n / |x| = \delta n\} \\ f_n &: D_n \to \{0,1\}^{\lfloor \rho n \rfloor \cdot (n+1)} \\ &(M,x) \to (M, M \cdot x) \end{aligned}$$

According to [Fischer and Stern, 1996], instances of the problem with weight δn very small or close to $n/2$ are not difficult. The instances of the collection SD with the weight δ near the Gilbert-Warshamov bound are believed to be unidirectional, although there is no proof in this sense. Thus we have the following assumption of intractability:

Intractability assumption Let ρ be in $]0,1[$. Then, for all δ in $]0,1/2[$, such that $H_2(\delta) < \rho$, the collection $SD(\rho,\delta)$ is strongly unidirectional.

Note that if $H_2(\lambda) = \rho$ and $\delta < \frac{\lambda}{2}$, the intractability of $SD(\rho, \delta)$ is a special case of decoding beyond half the minimum distance. Thus, the assumption becomes stronger than the usual assumptions for the decoding problem [Goldreich et al., 1993].

Cryptographic applications based on the complexity of known problems have been extensively studied and implemented in the last decades. Intractability assumptions are a natural step in building such applications. At the current state of knowledge in complexity theory, one cannot hope to build such algorithms without any intractability assumptions [Goldreich, 2001, p. 19].

3. Syndrome-Fortuna

The purpose of this work is to develop a random number generator and implement it inside the Linux kernel. The generator has its own scheme for entropy collection and the generating function is based on the intractability of the syndrome decoding problem. We will show that the proposed generator applies good grounds of security and is based on sound mathematical principles.

The generator was designed with independent functions of entropy collection and random values generation. Each one will be addressed separetely.

3.1. Fortuna: Entropy collection

[Ferguson and Schneier, 2003] proposed a cryptographically secure random number generator called Fortuna. The main contribution of Fortuna is the events collection system. It eliminated the need of entropy estimators, used until then in most of the generators we are aware of. Entropy estimation is commonly used to ensure that the generator state – or *seed* – is catastrophically updated, i.e., with sufficient amount of randomness. This should prevent potential adversaries who, for some reason, already know the seed, to iterate over all the possible new states and maintain control over the generator. If the possible new states are too many, it will not be feasible to try all of them.

3.1.1. Entropy accumulator

The Fortuna entropy accumulator captures data from various sources of entropy and uses them to update the seed of the generator. Its architecture, as we shall see, keeps the system secure even if an adversary controls some of the sources.

The captured random events must be *uniform* and *cyclically* distributed across n pools of the generator, as shown in figure 2. This distribution will ensure an even spread of entropy among pools, which will allow seed updates with successively larger amounts of randomness.

Each pool can receive, in theory, unlimited random data. To implement this the data is compressed incrementally, using the SHA 256 hash function, thus maintaining pools with constant size of 256 bits.

The generator state will be updated when the P_0 pool accumulate sufficient random data. A variable *counter* keeps track of how often the seed has been updated. This counter determines which pools will be used in the next update. A pool P_i will be used if, and only if, 2^i divides *counter*.

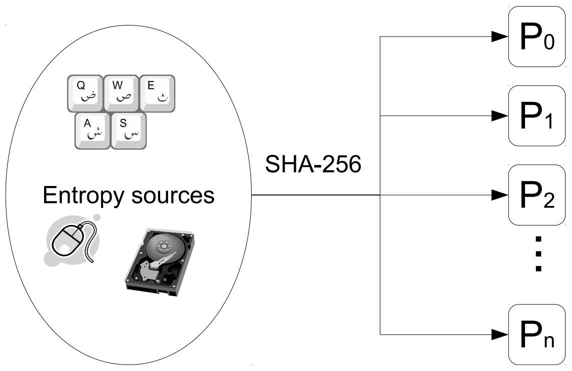

Figure 2. Entropy allocation among *n* pools. Data is compressed using the SHA 256 hash function.

Counter	Used pools
1	P0
2	P0, P1
3	P0
4	P0, P1, P2
5	P0
6	P0, P1
7	P0
8	P0, P1, P2, P3

Table 1. Used pools in the 8 first updates of Fortuna generator

Table 1 shows the update strategy of the generator. As we can see, the higher the number of the pool, less it is used in the update of the generator and, therefore, the greater the amount of accumulated entropy. This allows the generator to adapt automatically to attacks. If the opponents have little control over the sources of randomness, they can not even predict the state of the pool P_0, and the generator will recover from a compromised state quickly, at the next *reseed*.

However, the opponent may control multiple sources of entropy. In this case he probably knows a lot about the pool P_0 and could reconstruct the state of the generator using the previous state and the outputs produced. But when P_1 is used in a *reseed*, it contains twice the amount of randomness of P_0. When P_2 is used, it will contain four times the amount of randomness of P_0, and so on. While there are some truly unpredictable sources, eventually one pool will collect enough randomness to defeat the opponents, regardless of how many fake events they can generate or how many events they know the system would recover. The speed of recovery will be proportional to the level of opponents control over the sources.

3.1.2. Initial estimate

The proposed scheme avoids the entropy estimate, as used in Linux. However it is still necessary to make an *initial* entropy estimate in order to determine the minimum number of events necessary to perform a catastrophic reseed. This estimate is calculated for the pool P_0 and determines when the generator state is updated.

To elaborate such, one should observe some aspects of the system as: the desired security level; the amount of space occupied by the events and the amount of entropy present in each event.

[Ferguson and Schneier, 2003] suggest a minimum of 512 bits for P_0, for a level of 128-bit security. The initial entropy estimation plays an important role on system security, but is mitigated by the fact that if the chosen value is too low, there will always be *reseeds* with higher amounts of entropy. If the chosen value is too high, a possible recovery from a compromise may take longer, but will inevitably occur.

3.2. Syndrome: Generating function

3.2.1. Construction of the generating function

We built a PRNG based on a hard instance of the syndrome decoding problem using the SD collection of functions (ρ, δ) defined in section 2.2. Initially, we show that the functions $f_n^{\rho,\delta}$ from the collection $SD(\rho, \delta)$ expand its inputs, ie, they have image sets greater than the respective domain sets.

The domain $D_n^{\rho,\delta}$ from $f_n^{\rho,\delta}$ consists of a matrix M of size $\lfloor \rho n \rfloor \times n$ and of vector x of size n and weight δn. So the size of the domain set is $2^{\lfloor \rho n \rfloor \cdot n} \cdot \binom{n}{\delta n}$. Yet, the image set is formed by the same matrix M of size $\lfloor \rho n \rfloor \times n$ and a vector $y = M \cdot x$ of size $\lfloor \rho n \rfloor$. Thus, its size is $2^{\lfloor \rho n \rfloor \cdot n} \cdot 2^{\lfloor \rho n \rfloor}$.

So, for the image set to be larger than the domain set, we need $2^{\lfloor \rho n \rfloor} > \binom{n}{\delta n}$. This condition is satisfied when $H_2(\delta) < \rho$ according to the Gilbert-Warshamov bound defined in section 2.2.1. That is, for a sufficiently large n and suitable δ and ρ, $f_n^{\rho,\delta}$ expands its entry into a linear amount of bits.

Given an instance with fixed parameters ρ and δ from $SD(\rho, \delta)$ collection, we can construct an iterative generating function $G_{\rho,\delta}$ from the one-way function $f_n^{\rho,\delta}$. For this, we use an algorithm A that returns a vector x of size n and weight δn from a random vector with $\log_2 \binom{n}{\delta n}$ bits. This algorithm is described in section 3.2.2. The generator $G_{\rho,\delta}$ is described in pseudocode in algorithm 1. Figure 3 illustrates its operation.

Algorithm 1 *syndrome* – Iterative generating function based on the syndrome decoding problem.

```
Input: $(M, x) \in D_n^{\rho,\delta}$
Output: Print bit sequence
1: $y \leftarrow M \cdot x$
2: Split $y$ into two vectors $y_1$ e $y_2$. The firs with $\lfloor \log_2 \binom{n}{\delta n} \rfloor$ bits and the second with the
   remaining bits.
3: print $y_2$
4: $x \leftarrow A(y_1)$
5: Go to: 1
```

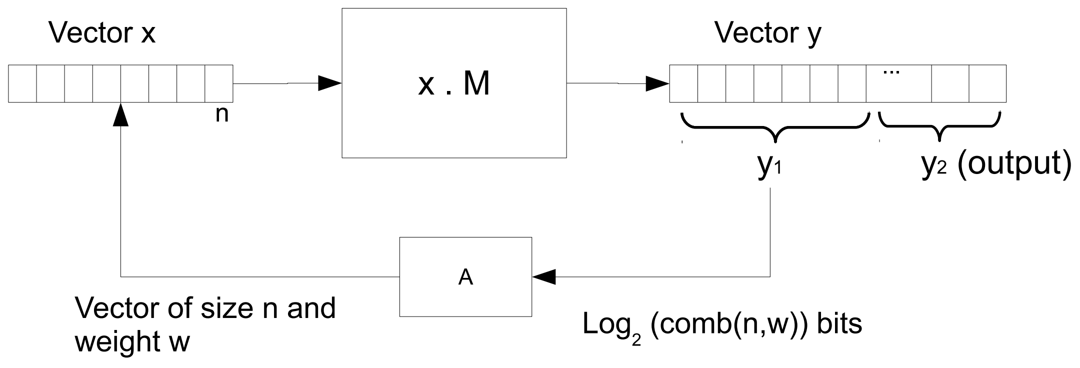

Figure 3. Syndrome generating function.

3.2.2. Generating words with given weight

As noted, the generator requires an algorithm to produce vectors with fixed weight. From each entry of size $\lfloor \log_2 \binom{n}{\delta n} \rfloor$, this algorithm must output a different vector $x \in \{0,1\}^n$ with weight $|x| = \delta n$. To build it, we use the lexicographic enumeration function shown by [Fischer and Stern, 1996].

To explain the working of the lexicographic enumeration, we will use *Pascal's Triangle*. It is formed by the binomial coefficients $\binom{n}{k}$ where n represents the row and k the column. The construction follows the Pascal's rule, which states:

$$\binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n}{k} \text{ for } 1 \leq k \leq n$$

Each triangle component $t(n,k) = \binom{n}{k}$ represents the number of existing words with size n and weight k. Here $t(n,k)$ equals the sum of two components immediately above $t(n-1,k)$ and $t(n-1,k-1)$. These components represent, respectively, the number of words of size n starting with a *0*-bit and starting with a *1*-bit.

This way, we can generate the output word from an index i by making an upward path in Pascal's Triangle. We start from the component $t(n,k)$ towards the top. When the index i is less than or equal to the up-*left* component $t(n-1,k)$, we generate a *0*-bit and move to this component. When the index is higher, we generate a *1*-bit and walk to the up-*right* component $t(n-1,k-1)$, decrementing the index at $t(n-1,k-1)$. The complete algorithm is described in pseudocode at the end of this section.

As an example, see Figure 4. We ilustrate a walk to generate a word of size $n = 4$ and weight $k = 2$, index $i = 2$.

The path begins at the component $t(4,2) = \binom{4}{2} = 6$. As $i = 2 \leq t(3,2) = 3$, the algorithm generates a 0-bit and walks to the component $t(3,2)$. Now, $i = 2 > t(2,2) = 1$, so the algorithm generates a 1 bit, updates the index $i \leftarrow i - t(2,2)$ and the path goes to the component $t(2,1)$. And so it goes until you reach the top of the triangle, forming the word $(0,1,0,1)$.

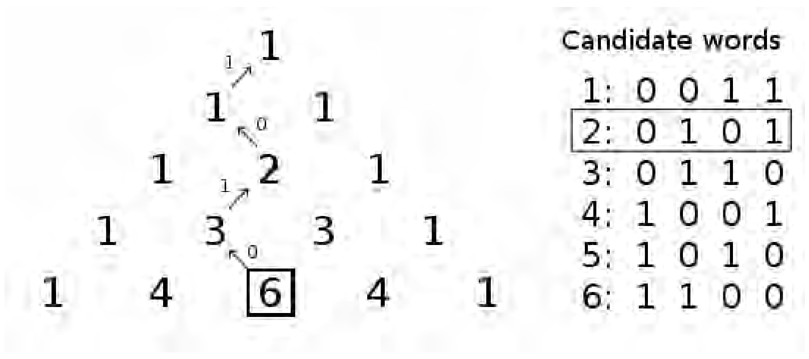

Figure 4. Walk in Pascal's Triangle to generate word of index $i = 2$ within a set of words of size 4 and weight 2

3.3. Combining Syndrome and Fortuna

The generating function constructed in 3.2.1 is based on the difficulty of finding a vector x with weight w from its syndrome, given a random matrix M. Thus, the only part of the function that actually makes up the internal state E, which must be kept secret, is the vector x. We will use, then, the entropy collection scheme to update the internal state. It should occur whenever the minimum amount of entropy is available in the P_0 pool. This way we ensure that the generating function receives the operating system randomness over time.

At each request for random numbers, a check is made whether there is enough entropy to update the internal state. This check is conditional on the minimum entropy in the pool P_0, as stipulated on the initial estimate. A minimum time lapse between reseeds is also respected. [Ferguson and Schneier, 2003] suggested *100ms* to prevent an adversary to make numerous requests and flush out the entropy of the pools. Figure 5 illustrates the complete workings of the generator.

The Syndrome-Fortuna update strategy preserves the one-way function direct cryptanalysis resistance. Only the seed, the vector x, is updated from time to time. Regardless of this update, any adversary that can reconstruct the internal state through the output vector y and the matrix M has managed to solve a problem currently considered computationally intractable.

Forward security is guaranteed by the feedback operation in which part of the result vector y is used to choose the new vector x. An adversary can, at time t, know the state of the generator $E_t = (x_t, M, P_t)$, where M is the matrix, x_t is the vector x at time t and P_t represents all the Fortuna Pools at time t. In this case, the opponent can simply find the last index value used in the lexicographic enumeration function $A^{-1}(x_t) = y1_{t-1}$. This value is part of the vector y_{t-1}, as can be seen in figure 5. From there, to find the value x_{t-1}, or the last output value $y2_{t-1}$ requires inverting the generating function.

The recovery to a safe state after a compromise – *backward security* – is guaranteed by the eventual update of vector x by the entropy collection system. An adversary who controls the state of the generator $E_t = (x_t, M, P_t)$ could possibly keep it until time $t + k$, when the accumulated entropy is sufficient for a catastrophic reseed. At this point the value of the vector x is changed by the hash function of Fortuna pools, as seen in figure 5. As long as the opponent has not enough knowledge about the entropy added to

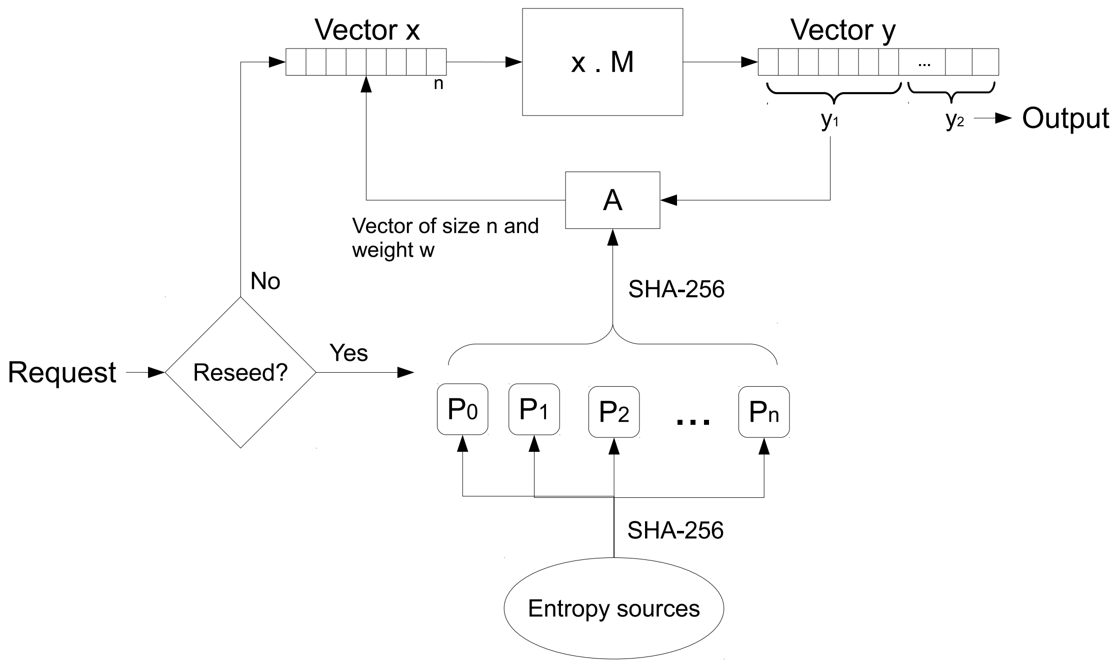

Figure 5. Syndrome-Fortuna operation. At each request is checked whether the pool P_0 has the minimum entropy and the time between *reseeds* is over 100ms. If so the algorithm performs a *reseed*, incrementing a counter c and choosing among the pools p_i that satisfies 2^i divides c. A SHA 256 is performed accross the chosen pools and the result is used to index a word of size n and weight w. Then, the generating function performs the multiplication between the chosen vector and the matrix M producing the syndrome vector y. Part of this vector is sent to the output and part is used as *feedback*, enabling the iterative generation of random data .

the pools, the new state E_{t+k+1} should be out of his control.

Ideally a system recovery should require 128 entropy bits [Ferguson and Schneier, 2003]. In the Fortuna entropy collection system this amount is multiplied by the number of pools, since the entropy is distributed. Thus, this amount rises to $128 * n$, where n is the number of pools. This value can be relatively high compared to the ideal; however, this is a constant factor, and ensures that the system will recover, at a future time.

In the above compromise case, considering the entropy input rate ω, the generator recovery time would be at most $k = (128 * n)/\omega$. Adversaries could try to deplete the system entropy while it is accumulated. They would need to promote frequent reseeds, emptying the pools before they contain enough entropy to defeat them. This could be done injecting false events on the pools through malicious drivers to keep the pool P_0 filled, allowing the real entropy flush. This attack is unlikely, given that the opponent would have to promote 2^n reseeds before the system collects $128 * n$ real entropy bits. However, to avoid any attempt a minimum time between state updates is used to prevent very frequent reseeds and the system entropy exhaustion.

3.4. Starting the generator

The initialization is a critical step for any random number generator that manages its own entropy. The problems related to lack of randomness at initialization time must be addressed according to each scenario. Therefore, it is beyond the scope of this paper to define specific strategies.

However, it should be noted that the implemented entropy accumulator allows the use of any source of randomness. Even a source of low quality, which can enter foreseeable data in the pools, has not the capacity to lower the system entropy. Other strategies, including the one used by the Linux kernel, estimate the collected amount of entropy. These approaches do not allow *questionable* sources to be used, since a miscalculation could lead to a security breach.

One good strategy to mitigate the problem of lack of entropy during boot is to simulate continuity between reboots. For the Syndrome-Fortuna generator a seed-file was implemented the same way as in Linux. The system writes a file with random data to disk during shutdown and startup. At the startup, the seed is read, fed to the generator and the output is written on disk before any request for random numbers. This prevents repeated states when sudden hangs occur. At startup, this seed-file is used to refresh the pools.

4. Implementation, analysis and results

4.1. Testing methodology

One way to evaluate a random number generator quality is assessing its output's statistical behavior. This analysis does not guarantee, in any way, the security of a cryptographic generator. However, it can reveal flaws on its design or implementation.

There are several libraries for statistical tests accepted by the scientific community. We used the libraries *SmallCrush* and *Crush* from *TestU01* library, developed by [L'Ecuyer and Simard, 2007]. The first one implements a set consisting of 10 tests and is recommended for a generator's initial assessment. The second library applies 32 tests with several configurations, evaluating a total of 2^{35} random bits.

To evaluate the generator performance, we compared it with the Blum-Blum-Shub generator, which has a simple construction, based on the integer factoring difficulty. We also compared to the Linux kernel 2.6.30 generator (LRNG). TestU01 library was used to measure the generators performance.

4.2. Statistical Results

The statistical tests results are presented in the shape of *p-values*, which indicate the likelihood of a sample Y present the sampled value y, considering true the null hypothesis H_0:

$$p = P(Y \geq y|H_0)$$

To illustrate this statistical approach, we evaluate a sample Y of 100 coin flips in which 80 times was randomly selected "heads" and 20 times "tails". In this case, the null hypothesis is the coin fairness and therefore Y is a cumulative binomial distribution. Thus, we have $p = P(Y \geq 80) = 5.6 * 10^{-10}$. The observed sample could occur with a fair coin with only $5.6 * 10^{-10}$ probability. This is not to tacitly reject the null hypothesis, but according to a defined demand level, we can consider the sample clearly failed the applied test. [L'Ecuyer and Simard, 2007] arbitrate as *clear failures* the *p-values* outside the range $[10^{-10}, 1 - 10^{-10}]$.

All generators tested: BBS, Syndrome-Fortuna and LRNG passed the statistical test libraries. All *p-values* were in the range $[10^{-3}, 1 - 10^{-3}]$, forbidding us to reject the null hypothesis. This means that, statistically, the generated values cannot be distinguished from truly random values.

4.3. Performance analysis

The Syndrome-Fortuna generator was evaluated through performance tests and compared to the BBS and LRNG generators. We used a computer with an Intel Pentium Dual Core T3400 2.16GHz with 2GB of RAM. We used loads of 100 bytes, 1 kB, 10kB, 100kB and 100kB intervals until complete 1MB of random data. Each generator had the generating time clocked 3 times for each load.

Syndrome-Fortuna and LRNG were assessed while compiled into the kernel. The BBS algorithm was used only as a benchmark for performance and was not implemented within the kernel, being assessed with TestU01 library.

To evaluate the performance diferences between generators built inside and outside the kernel, we did tests with an implementation of Syndrome-Fortuna compiled in user-space. The results were statistically indistinguishable from the results when the algorithm was implemented inside the kernel. This does not necessarily imply that the same would happen to the BBS algorithm, the only algorithm not implemented inside the kernel. But for the purposes of this paper, we consider it satisfactory for the comparision of the BBS, compiled in user space, with Syndrome-Fortuna and LRNG, built inside the kernel.

The average speed of each generator, with the respective deviation, are shown in table 2. The generators behavior for the different loads are summarized in figure 6.

Generator	**Speed (em kB/s)**
LRNG	2664,0 $\pm$ 818,9
Syndrome-Fortuna	197,1 $\pm$ 58,2
BBS	31,4 $\pm$ 6,4

Table 2. Generators LRNG, Syndrome-Fortuna and BBS performance measurement.

As expected, Syndrome-Fortuna underperforms the current Linux generator, which is highly optimized for performance and does not have formal security proof. When compared with the BBS generator, which has similar formal security features, the Syndrome-Fortuna performance is 6.3 times higher.

5. Concluding remarks

During this research we studied several types of random number generators, statistical and cryptographic. We conducted a detailed investigation of the Linux kernel generator, and proposed and implemented a new generator based on two existing approaches.

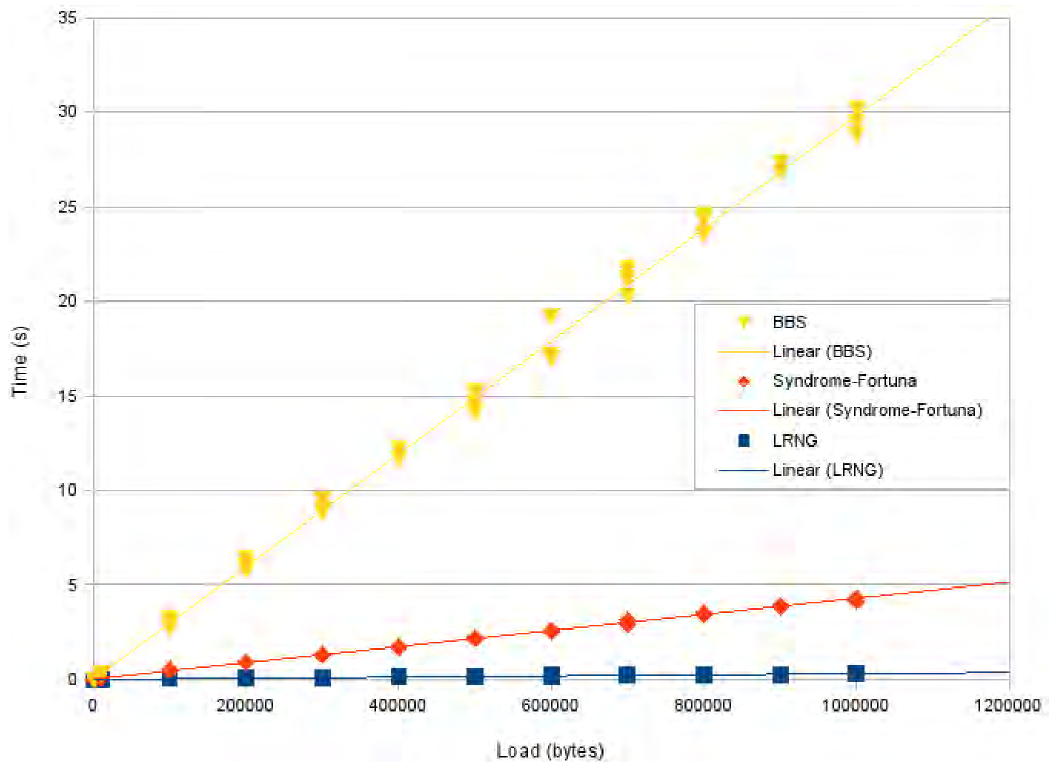

Figure 6. Performance of random generators: Linux (LRNG) Syndrome-Fortuna and Blum-Blum-Shub (BBS).

5.1. Conclusions from our research

Random number generators are an important part of the complex set of protocols and applications responsible for ensuring information security. In a scenario of rapid change, when the computing reach unexplored places, and new users, the framework for cryptographic applications must adapt to provide the desired security.

For random number generators, this means adapting to new sources of entropy, new forms of operation. It is not difficult to imagine scenarios where a keyboard, mouse and hard drive are less present, imposing restrictions on the randomness of the systems. The strategy we implemented for entropy collection is in line with this need. It does not require estimations and can use any entropy sources, even the ones with questionable randomnness.

Conversely, as noted in its operation, the Linux random number generator faces a difficulty to adapt to new entropy sources. All of them must pass through a heuristic that, if inaccurate, can lead to a generator low entropy state. In a running Linux Ubuntu 9.10, we observed the entropy collection only from the keyboard, mouse and hard drive, while a series of possibly good entropy sources were available, such as wireless network cards, software interrupts, etc.

As for the random values generation, per se, the implementation applies solid mathematical principles derived from the theory of error-correcting codes, and predicting them can be considered as difficult as the syndrome decoding problem, which is NP-complete.

The proposed algorithm can be considered for use in various scenarios, especially on diskless servers, or in cloud-computing scenarios, where the usual randomness sources are not present. We believe that the generator implements a satisfying trade-off, providing bits with good statistical properties, solid security and reasonable speeds

5.2. Future work

We believe that the Syndrome-Fortuna time and memory performance can be improved considerably by changing the generating function "A", shown in figure 5. We note that much of the processing and, clearly, most of the memory costs are related to the problem of obtaining the vector x of size n and weight w from a random index i. The approach used in the work of Augot et al. [Augot et al., 2005] could reduce drastically these costs. Generator specific parameters should be studied and modified to allow this use.

As shown, the entropy collection strategy allows the use of new randomness sources, independent of detailed entropy studies. A natural extension of this work is to identify these sources and promote their interconnection with the Linux kernel entropy collection system.

References

Augot, D., Finiasz, M., and Sendrier, N. (2005). A family of fast syndrome based cryptographic hash functions. In Dawson, E. and Vaudenay, S., editors, *Mycrypt 2005*, volume 3715, pages 64–83. Springer.

Berlekamp, E., McEliece, R., and Van Tilborg, H. (1978). On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386.

Chabaud, F. (1994). On the security of some cryptosystems based on error-correcting codes. In *EUROCRYPT*, pages 131–139.

Ferguson, N. and Schneier, B. (2003). *Practical Cryptography*. Wiley & Sons.

Fischer, J.-B. and Stern, J. (1996). An efficient pseudo-random generator provably as secure as syndrome decoding. In *EUROCRYPT*, pages 245–255.

Goldberg, I. and Wagner, D. (1996). Randomness and the Netscape browser. *Dr. Dobb's Journal of Software Tools*, 21(1):66, 68–70.

Goldreich, O. (2001). *Foundations of Cryptography. Volume I: Basic Tools.* Cambridge University Press, Cambridge, England.

Goldreich, O., Krawczyk, H., and Michael, L. (1993). On the existence of pseudorandom generators. *SIAM J. Computing*, 22(6):1163–1175.

Impagliazzo, R., Levin, L., and Luby, M. (1989). Pseudorandom generation from one-way functions. In *Proc. 21st Ann. ACM Symp. on Theory of Computing*, pages 12–24.

L'Ecuyer, P. and Simard, R. (2007). Testu01: A c library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4).