

# On Linux Random Number Generator

A thesis submitted in partial fulfillment  
of the requirements for the degree of

Master of Science

by

**Tzachy Reinman**

supervised by

Prof. Dahlia Malkhi

School of Engineering and Computer Science  
The Hebrew University of Jerusalem  
Jerusalem , Israel

Elul 5765

September 2005



# Acknowledgments

I am very grateful to G-d, the Creator and the Guardian, to whom I owe my very existence.

I would like to thank Prof. Dahlia Malkhi for her guidance and support in this research.

I wish to express my deepest gratitude to Zvika Gutterman. I am grateful to him for his guidance and for investing unlimited time and efforts during the study.

I would also like to thank the members of DANSS Lab for their encouragement.

Special thanks to Yaron Sella, Noa Bar-Yosef, Yair Gheva, and my father for proof-reading previous versions and drafts of this work.

Thanks also to Tom and Rafa from the System group.

Finally, I wish to thank my parents, Batya and Meir, for their support and love.

Last but not least, many thanks to my dearest wife Osnat, and our kids Be'eri and Aluma. They sacrificed for this work more than anyone else. It could not be completed without their patience and love.



# Abstract

Linux is probably the most popular open source project installed on millions of computers around the world. The Linux random number generator is an entropy based intra operating system pseudo-random-number-generator which is part of the core of all Linux distributions. The Linux generator is used for almost any security protocol including TLS key generation, TCP sequence number, file system and email encryption. Despite the fact that the Linux generator is part of the kernel open source project and its source code is less than 2500 lines of code, the algorithm is not documented and hundreds of code patches during the last five years make it much more cumbersome. Using dynamic and static reverse engineering this thesis presents an analysis of the Linux generator and a few cryptographic flaws. Our analysis included developing a user mode simulator for the Linux generator.

# Table of Contents

<b>Acknowledgments</b>	<b>1</b>
<b>Abstract</b>	<b>3</b>
<b>1 Introduction</b>	<b>7</b>
<b>2 Pseudo Random Number Generators</b>	<b>9</b>
2.1 The Importance of Random Numbers . . . . .	9
2.2 History . . . . .	11
2.3 PRNGs for Simulations, Numerical Analysis, Sampling and Games . . . . .	12
2.3.1 Linear Congruential Generator . . . . .	12
2.3.2 Linear Feedback Shift Register . . . . .	13
2.3.3 Generalized Feedback Shift Register . . . . .	14
2.3.4 Twisted Generalized Feedback Shift Register . . . . .	14
2.4 Cryptographic PRNGs . . . . .	16
2.4.1 Shamir . . . . .	16
2.4.2 Blum-Blum-Shub . . . . .	17
<b>3 Attacks on PRNGs</b>	<b>18</b>
3.1 Attacks Based on Linear System Solution . . . . .	18
3.2 Berlekamp-Massey Algorithm . . . . .	20
3.3 Wagner and Goldberg Attack on the Netscape Browser . . . . .	22
3.4 Gutterman and Malkhi Attack on Java Session-id Generation . . . . .	23

---

<b>4</b>	<b>Related Work—OS-based PRNGs</b>	<b>25</b>
<b>5</b>	<b>Linux Random Number Generator Structure</b>	<b>28</b>
5.1	Initialization . . . . .	31
5.2	Collecting Entropy . . . . .	32
5.3	Adding Entropy . . . . .	33
5.4	Estimating the Entropy Amount . . . . .	34
5.5	Extracting Random Bits . . . . .	36
<b>6</b>	<b>Attacks, Weaknesses and Security Flaws</b>	<b>39</b>
6.1	Denial of Service . . . . .	39
6.2	Predictability of /dev/urandom . . . . .	40
6.3	Uninitialized Memory Read . . . . .	40
6.4	Guessable Passwords . . . . .	41
6.5	Creating Noise that Directly Affects the LRNG Output . . . . .	42
<b>7</b>	<b>Discussion and Conclusions</b>	<b>43</b>
	<b>Bibliography</b>	<b>47</b>

# List of Figures

2.1	Diffie-Hellman Key Exchange . . . . .	10
2.2	Linear Feedback Shift Register . . . . .	13
2.3	Generalized Feedback Shift Register . . . . .	15
2.4	Twisted Generalized Feedback Shift Register . . . . .	15
2.5	Blum-Blum-Shub PRNG . . . . .	17
3.1	Berlekamp-Massey Algorithm . . . . .	21
3.2	Berlekamp-Massey Example . . . . .	22
3.3	The Netscape Seeding Process . . . . .	22
4.1	Comparison of OS-Based PRNGs . . . . .	27
5.1	Linux Random Number Generator . . . . .	29
5.2	LRNG FSR . . . . .	35
5.3	Extracting Algorithm Pseudo Code . . . . .	37
5.4	Folding Operation . . . . .	38
6.1	Uninitialized Memory Use when Transferring Entropy Between Pools	41

# Chapter 1

## Introduction

In this thesis we study the Linux Random Number Generator (LRNG), an intra operating system based random number generator which plays a crucial role in almost any cryptographic protocol in Linux.

The Linux kernel is an open source project developed in the last 15 years by group of developers led by Linus Torvalds. The kernel is the common element in all various Linux distributions.

Despite the fact that the Linux Random Number Generator (LRNG) is part of an open source project and its source code is less than 2500 lines of code, the algorithm is not documented and the hundreds of code patches during the last five years make it much more cumbersome. Hence, we have used both static analysis of the Linux kernel source and dynamic tracing to reverse engineer the generator algorithm. We implemented a user mode simulator of the LRNG as part of our analysis. It can be downloaded from the author's web page at <http://www.cs.huji.ac.il/~reinman>.

LRNG usage is divided into internal kernel functionalities using random bits and the application programming interface. The Linux kernel uses random data for various purposes: generating random unique identifiers, computing TCP sequence numbers, producing passwords, generating SSL private keys and additional intra-kernel functions. Within the kernel, the interface is the function `void get_random_bytes(void *buf, int nbytes)`.

The user interface is through two device drivers named `/dev/random` and

`/dev/urandom`. Both devices let users read from them pseudo random bits. The difference between the two resides in the level of security and the delay. The first device (`/dev/random`) outputs extremely<sup>1</sup> secure bits and may block the user till such bits exist in the system. The second device (`/dev/urandom`) outputs less secure bits but its output is never blocked. Section 5.4 explains the difference between the two devices.

The blocking also presents an immediate denial-of-service attack which we present in Chapter 6 where an adversary with user privileges can deny other users on the same machine from access to random bits. In certain cases this attack can be mounted remotely, without a local account.

At a high level, the LRNG can be described as three asynchronous elements. At first, occurrence of intra operating systems events “push” bits into the generator pool. Later these bits are xored with the pool content. When bits are read from the generator, three consecutive SHA-1<sup>2</sup> operations are preformed. The first two include feedback into the pool and part of the last hash is output to the consumer.

“Randomness” originated from intra operating system events is collected as two 32-bit words. Half of the bits measure the time of the event and the second half is the event value, which is usually an encoding of the pressed key, mouse move or drive access. LRNG design goal is to output only non-predictable bits which should originate from non-predictable events. For enforcing that, the pool holds a counter for counting the non-predictable bits, which is calculated as a function of the frequencies of the different events. LRNG names this value *entropy* (see Section 5.4 for the exact definition), which is completely different than the classical entropy definition by Shannon [1].

The rest of this thesis is organized as follows. We start with a survey of pseudo random number generators and some known attacks in Chapter 2 and Chapter 3, respectively. We describe some operating system’s PRNG in Chapter 4. An analysis of the LRNG algorithm is presented in Chapter 5. The various attacks are described in Chapter 6, and our conclusions are in Chapter 7.

---

<sup>1</sup>This is according to the LRNG designer, our discussion follows in Chapter 6.

<sup>2</sup>The latest attacks on SHA-1 did not seem relevant in the Linux generator.

## Chapter 2

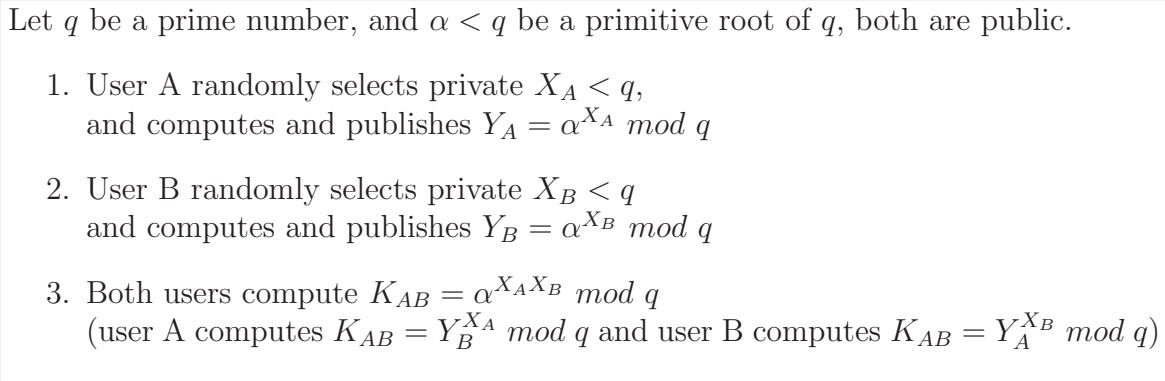
# Pseudo Random Number Generators

Computers are deterministic machines, and as such, they cannot produce truly random numbers. As John Von-Neumann said: “Anyone who considers arithmetic methods of producing random digits is, of course, in a state of sin”. From this reason we refer to *Pseudo Random Number Generators* (PRNG). *Pseudo* implies that a PRNG does not produce truly random numbers, but numbers that are *random enough* for a specific practical use. In this chapter we first (Section 2.1) demonstrate the importance of random numbers and show what happens when PRNGs are used wrongly. Section 2.2 describes the evolution of PRNGs and distinguishes between PRNGs for statistical or simulations uses and PRNGs for cryptography purposes. In Section 2.3 and Section 2.4 we present some PRNG algorithms of the two kinds, respectively.

### 2.1 The Importance of Random Numbers

Random number generators are one of the most important building blocks of many cryptographic paradigms and security protocols. TLS [2], PGP [3, 4, 5] and Secret Sharing [6], are just few examples. We demonstrate the importance of random numbers in crypto-systems by Diffie-Hellman key exchange algorithm.

A public-key cryptography, as discovered by Whitfield Diffie and Martin E. Hellman [7], makes use of random numbers. The first published public-key algorithm, known as Diffie-Hellman key exchange, depends on the difficulty of the discrete logarithm problem and counts on the ability to select numbers at random. The algorithm is described in Figure 2.1.



**Figure 2.1:** Diffie-Hellman Key Exchange Algorithm.

Only the followings ingredients are public:  $q$ ,  $a$ ,  $Y_A$  and  $Y_B$ . If an adversary wishes to reveal  $K_{AB}$ , he must compute (without loss of generality)  $K_{AB} = Y_A^{X_B} \bmod q = Y_A^{\log_\alpha Y_B} \bmod q$ . Since the discrete logarithm problem is difficult, this is infeasible (assuming  $q$  is large enough). This is true only as long as  $X_B$  is unknown. If  $X_B$  is not chosen at random, or if its randomness is weak, the adversary can guess or determine it, and then he can compute  $K_{AB}$  as user B can, without solving the logarithm.

Using PRNG correctly is a critical factor in many applications. Schneier [8] gives an example of an incorrect use of PRNG in a casino.

The PRNG used in the casino is constructed of a seed quantity and a mixing function that produces the output. By default, whenever the PRNG is initialized, a hard-coded seed is used. This is mostly used for debugging purposes. To use the PRNG in a realistic setting, the operator should execute a “set-seed” procedure, which computes a new seed, usually by using the machine’s time-of-day.

What apparently happened is that for some reason, the machine was shut-down at some point during the day or night. When it resumed, it was re-seeded with the

hard-coded seed, i.e., with the same seed every day.

## 2.2 History

Random number generation is as old as computer science and pioneers such as Turing and Von-Neumann had already begun studying the generation of random sequences (Knuth's introduction to random number generators [9] includes many additional historical notes). Computers' first need of random numbers was for simulations and numerical computations such as Monte Carlo calculations [10]. It is important to note that numerical analysis' need for randomness is different than that of cryptographic applications. Prediction is often a requirement when repeating a numeric experiment and a hazard when choosing values for RSA keys.

James Reeds [11] enumerate two distinguished standards of randomness.

- The *usual statistical standards* states that a sequence of numbers, that cannot be discriminated from a sequence of independent uniform deviates from the unit interval, is considered random. The tests for this comparison are statistical analysis, detailed and discussed in [9]. PRNGs that are acceptable by these standards are suitable for simulations, sampling, games, and similar applications.
- The *cryptography standards* has to do with predictability. It is less important whether the sequence is uniformly distributed, but it is essential that knowing part of the sequence does not contribute any knowledge about other parts. This requirement is called *unpredictability*.

RFC 1750 [12] points out that cryptography usages of random numbers is also characterized by the ability of adversaries to guess the random value many times (to simulations where there is no adversary, or in games where the number of tries is very limited). Recall the casino story. If the machine was seeded as expected, using the time of day, the gambler would not have beaten the game. But from a cryptographic point of view, this is not enough— an opponent can try many times and

finally guess the time of day correctly. In addition, computer clocks provide very little unpredictability, due to the fact that only the lower bits are changed frequently.

The difference between these two standards is demonstrated later, in Chapter 6, when we show how PRNGs such as linear congruential generator, that are acceptable by the first standards, are attackable from the point of view of cryptography standards.

While concerned with the security of PRNG the following definition is common:

**Definition 1** *A PRNG which cannot be distinguished from RNG by a polynomial time algorithm is a Secure PRNG (SPRNG).*

Tens of random number generators exists (e.g., [13], [14], [15], [16], [17], [18], [19], [20]). Shamir was first to provide SPRNG [16] while Blum-Blum-Shub [17] and many other PRNGs followed.

## 2.3 PRNGs for Simulations, Numerical Analysis, Sampling and Games

The following PRNGs were developed originally for simulations, numerical analysis, sampling and games. They have been applied for cryptography. Analysis of cryptographic use of some of them follows in the next chapter.

### 2.3.1 Linear Congruential Generator

Linear congruential generator (LCG) [13] is the most widely used technique for non-cryptographic use. The output sequence of random numbers is produced in the following manner.

$$X_{i+1} = aX_i + c(\text{mod } n)$$

where  $a$ ,  $c$ , and  $n$  are constants, and  $X_0$  is the seed. Selecting them must be done carefully, to ensure a maximal period of  $n - 1$ . For example, if  $a = 1$  and  $c = 0$ , the output will be a sequence of repeatedly  $X_0$ . [9] (§ 3.2.1.1) discusses the selection of these values in depth.

### 2.3.2 Linear Feedback Shift Register

Linear feedback shift register (LFSR) is composed of a register of  $n$  memory elements, each of them is capable of storing one bit, and having one input and one output; and a clock which controls the *shift* operation (the flow of data between these elements) and the *feedback* operation.

At each time unit the content of element 0 is output, the content of each element  $i$  is moved to element  $i - 1$ , and the new content of element  $n - 1$  is the output of the feedback function.

The feedback function is *xor* of the content of a fixed subset of the register elements. This subset (also called a *tap sequence*) is represented by a polynomial of the form  $\sum c_i x^i + 1$  where  $1 \leq i \leq n$  and  $c_i \in \{0, 1\}$  ( $c_i = 1$  if the  $i^{\text{th}}$  element is in the tap sequence).

Figure 2.2 depicts a LFSR, with  $n = 5$ , and the polynomial  $x^4 + x^3 + x^2 + 1$ .

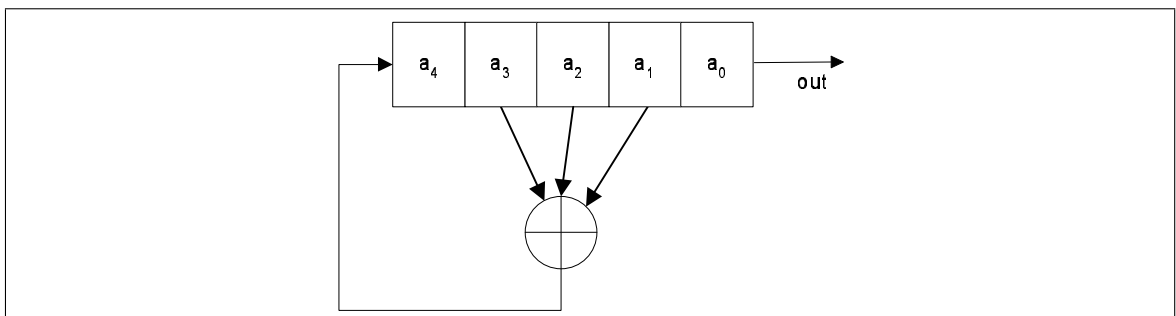


Figure 2.2: LFSR.

If the first element is used as input to the feedback function (i.e.,  $c_0 = 1$ ), the LFSR is said to be *non-singular*. If the LFSR is non-singular and the polynomial is a primitive polynomial, and the initial content of the register is not all zero, then the LFSR produces output with the maximum possible period  $2^n - 1$ . [21] (§ 16.2) lists of primitive polynomials modulo 2.

Schneier [21] (§ 16.4) lists more than a dozen PRNGs that are constructed as combinations of different LFSRs.

### 2.3.3 Generalized Feedback Shift Register

Generalized feedback shift register (GFSR) [15] is a refinement of LFSR. It is non-singular, the polynomial is primitive, and its degree is 3 (a trinomial). Among GFSR's advantages are: better distribution, long period, and speed.

**Definition 2** A series  $a_i \in \{0, 1\}^w, i \in N$  is GFSR based on primitive polynomial  $x^p + x^q + 1$  if and only if

$$a_i = a_{i-p+q} \oplus a_{i-p} \quad ; \quad i = p, p + 1, \dots$$

This definition is very similar to LFSR. The main differences between GFSR and LFSR are:

- The memory element in GFSR has a word of  $w$  bits (in LFSR it has one bit).
- The period length of GFSR output does depend on the initial seed. This seed is produced from a non-zero sequence of  $p$  bits, which is extended in the following way. First, the sequence is extended to  $2^w$  bits length, using the trinomial, in the LFSR fashion. Then, the extended sequence is set into  $j \leq p$  columns, with a judiciously selected delay. This forms a sequence of  $2^w$   $w$ -bit words. The first  $p$  words are the initial seed  $a_0$  to  $a_{p-1}$ . The mechanism of creating the seed is explained in detail in [15].

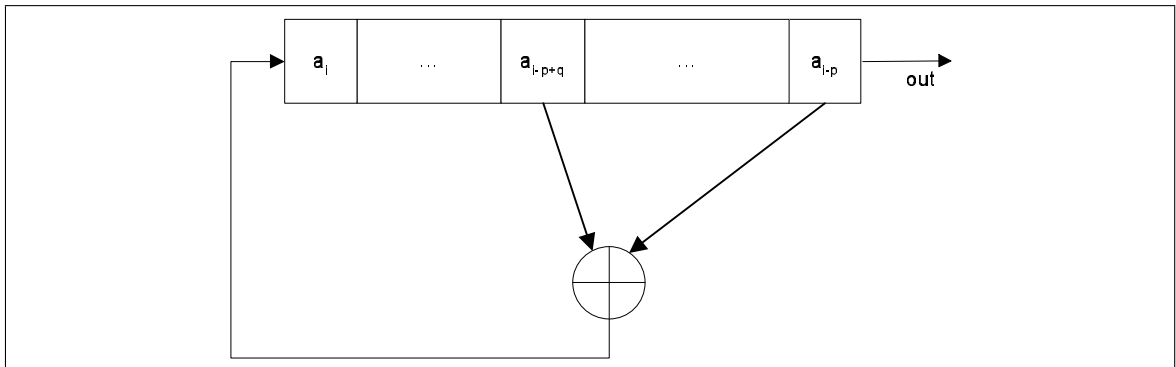
GFSR has three main properties which are relevant from the security aspect. The first is the long cycle, the second is the linearity of elements and the third has to do with the fact that cycle length is a factor of the seed.

Figure 2.3 depicts a GFSR.

### 2.3.4 Twisted Generalized Feedback Shift Register

Twisted generalized feedback shift register (TGFSR) [18, 19] improves the cycle length from  $2^p - 1$  to  $2^{pw} - 1$ , and cuts off the dependency in the initial seed. In addition, the polynomial may be of any degree, not necessarily 3. In this sense TGFSR is a generalization of GFSR.

## 2.3 PRNGs for Simulations, Numerical Analysis, Sampling and Games 15



**Figure 2.3:** GFSR. Based on a primitive polynomial  $x^p + x^q + 1$ .

**Definition 3** A series  $a_i \in \{0, 1\}^w, i \in \mathbb{N}$  and a matrix  $A_{w \times w}$  are a TGFSR based on primitive polynomial  $x^p + x^q + 1$  if and only if

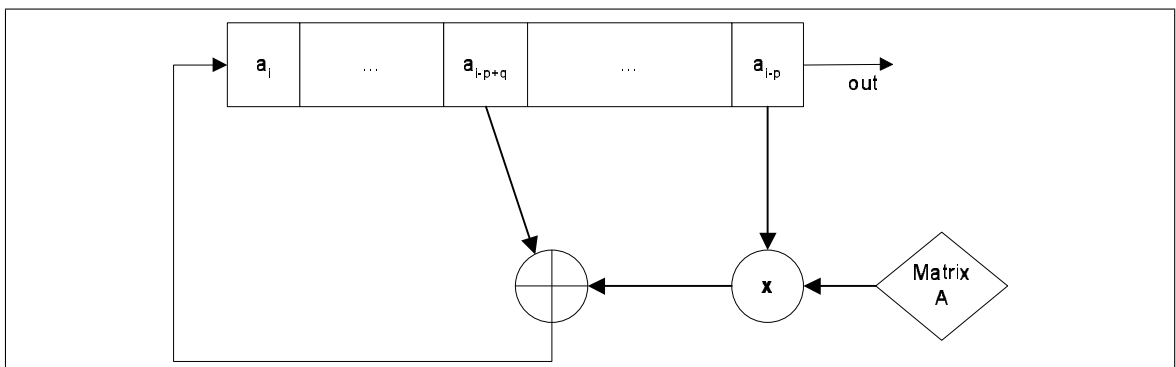
$$a_i = a_{i-p+q} \oplus a_{i-p}A \quad ; \quad i = p, p+1, \dots$$

The initial seed  $a_0$  to  $a_{p-1}$  are not all zero.

The product  $a_{i-p}A$  can be implemented with *shift-right* and *xor* operations, see [18] for details.

*Mersene Twister* [20] is a commonly used improved variant of TGFSR, with a long period of  $2^{19937} - 1$ . It is very efficient, both in time and place.

Figure 2.4 depicts a TGFSR.



**Figure 2.4:** TGFSR. Based on a primitive polynomial  $x^p + x^q + 1$ .

## 2.4 Cryptographic PRNGs

Following the unsuitability of the so called “statistical” PRNGs for cryptographic purposes, special PRNGs, intended for cryptography uses, were developed. We present here two cryptographically strong PRNGs, the first [16] is mainly of theoretical interest, and the second [17] is widely used in practice.

### 2.4.1 Shamir

Shamir [16] was the first to publish a PRNG that is proved to be cryptographically strong in the sense of unpredictability.

Shamir’s scheme is based on the RSA public-key encryption function. Its strength is proved in the same level of certainty of some known computational tasks, such as the existence of one-way functions, the difficulty of factoring integers and the assumption  $P \neq NP$ . On the other hand, since it uses modular exponentiation of huge numbers, it is not suitable for practical applications.

Shamir’s PRNG consists of the following numbers:

- $N = pq$ , where  $p$  and  $q$  are secret large prime numbers
- A seed  $S$
- A sequence of keys  $K_1, K_2, \dots$ , such that  $\varphi(N)$  and all the  $K_i$ s are pairwise relatively prime

The random numbers sequence is:

$$R_1 = S^{1/K_1}(\text{mod } N), R_2 = S^{1/K_2}(\text{mod } N), \dots$$

The computation of the  $K_i^{\text{th}}$  root of  $S$ ,  $R_i = S^{1/K_i}(\text{mod } N)$ , is easy given  $p$  and  $q$  and difficult if they are unknown.

Shamir proves that computing  $R_1$  given  $N$ ,  $S$  and  $R_2, \dots, R_l$  is as difficult as computing  $R_1$  given only  $N$  and  $S$ . This means that knowing part of the random numbers sequence does not contribute knowledge about other parts of the sequence.

It is also proved that the security of this scheme is equivalent to the security of the RSA cryptosystem.

### 2.4.2 Blum-Blum-Shub

Blum-Blum-Shub (BBS) [17] is a generator based on quadratic residues. It is the simplest and most widely used crypto-orientation PRNG. BBS is appropriate only for cryptography and not for simulation, because it is not very fast.

BBS parameters are two large prime numbers  $p$  and  $q$  such that  $p = q = 3 \pmod{4}$ .  $N = pq$  is called the Blum integer.

The sequence is produced as follows. First, choose a random number  $s$  which is relatively prime to  $N$ .  $s$  must remain secret. The initial seed is  $S_0 = s^2 \pmod{N}$ .

The random numbers sequence is the lowest  $k$  bits of  $S_i = S_{i-1}^2 \pmod{N}$ , where  $k \leq \log_2 \log_2(S_i)$ . Usually the least significant bit is taken, i.e.,  $k = 1$ .

It is worthwhile to note that any  $S_i$  can be directly calculated from the initial seed  $S_i = S_0^{2^i \pmod{\varphi(N)}} \pmod{N}$  where  $\varphi(N) = (p-1)(q-1)$ . This means that there is no need to save all the previous values.

The security of BBS (as well as the security of Shamir's scheme) is based on the difficulty of factoring  $N$ .

Figure 2.5 summarizes the BBS algorithm.

$$\begin{aligned}
 &S_0 = s^2 \pmod{N} \\
 &\text{for } i = 1 \text{ to } \infty \\
 &\quad S_i = S_{i-1}^2 \pmod{N} \\
 &\quad B_i = S_i \pmod{2} \\
 &\text{end for}
 \end{aligned}$$

**Figure 2.5:** BBS.  $N$  is the modulo,  $s$  is the secret,  $S_0$  is the initial seed,  $B_i$  are the outputs.

# Chapter 3

## Attacks on PRNGs

Some PRNGs were proved to be weak and easily predictable. Lehmer [13] linear congruential generator was first broken by Reeds [11] and then proved by Boyar [22, 23] to be predicted in  $O(\log n)$  iterations. The Linear Feedback Shift Register (LFSR) is predictable using the Berlekamp-Massey [24, 25] algorithm for finding the primitive polynomial.

Weak random values may result in an adversary ability to break the cipher or security cryptosystem. Breaking Netscape SSL [26] and the prediction of Java session-ids [27] demonstrate the vulnerability of cryptosystems which use weak random numbers.

In this chapter we present some known attacks on PRNGs.

### 3.1 Attacks Based on Linear System Solution

Reeds [11] was the first to describe a prediction attack on LCG, using a linear system solution. He demonstrated it on a few numerical examples. Boyar [22, 23] generalized the algorithm, and showed how to solve the linear system. In addition, she proved that these methods can be applied to break any PRNG of the form  $X_n = \sum_{j=1}^k \alpha_j \phi_j(X_0, X_1, \dots, X_{n-1}) \pmod{m}$  where the functions  $\phi_j$  are known, and the coefficients  $\alpha_j$  and the modulo  $m$  are unknown. If the functions  $\phi_j$  (without the modulo reduction) are computable in time polynomial in  $\log m$  and  $k$ , the time of

predicting the next element is also polynomial in  $\log m$  and  $k$ .

We present here the basic prediction attack of Reeds.

Recall Section 2.3.1, the LCG formula is  $X_{i+1} = aX_i + c(\text{mod } n)$  where  $a$ ,  $c$ , and  $n$  are constants, and  $X_0$  is the seed.

Reeds' attack input is 4 consecutive numbers  $X_j$  to  $X_{j+3}$ , its output is the constant parameters  $a$ ,  $c$ , and  $n$ , which enable to predict all the sequence. The attack is executed by solving the following equations for the unknowns  $a$ ,  $c$ , and  $n$ .

$$\begin{aligned} X_{j+1} &= aX_j + c(\text{mod } n) \\ X_{j+2} &= aX_{j+1} + c(\text{mod } n) \\ X_{j+3} &= aX_{j+2} + c(\text{mod } n) \end{aligned}$$

A similar attack, based on a linear system solution, can be applied to LFSR. Recall Section 2.3.2, LFSR is based on a polynomial of the form  $\sum c_i x^i + 1$ . Given a sequence produced by the LFSR, revealing the  $c_i$ s enables predicting the subsequent bits. If the LFSR is of length  $n$ , then a sequence of  $2n$  bits is enough to create a linear system that can be solved for the  $c_i$ s.

For example, let the known sequence of an LFSR of length 4 be  $S = 011001000111101\dots$ . We denote the first bit  $S_0$ , the second  $S_1$  and so on. Since the LFSR length is  $n = 4$ , each 4 bits determine the next bit: Bits  $S_0$  to  $S_3$  determines bit  $S_4$ ,  $S_1$  to  $S_4$  determines  $S_5$ , and so on. Below is a demonstration of this process for the first  $2n = 8$  bits.

<i>line</i>	$c_3$	$c_2$	$c_1$	$c_0$	
1	$S_3$	$S_2$	$S_1$	$S_0$	$\Rightarrow S_4$
	0	1	1	0	$\Rightarrow 0$
2	$S_4$	$S_3$	$S_2$	$S_1$	$\Rightarrow S_5$
	0	0	1	1	$\Rightarrow 1$
3	$S_5$	$S_4$	$S_3$	$S_2$	$\Rightarrow S_6$
	1	0	0	1	$\Rightarrow 0$
4	$S_6$	$S_5$	$S_4$	$S_3$	$\Rightarrow S_7$
	0	1	0	0	$\Rightarrow 0$

This is a  $4 \times 5$  matrix, whose variables are  $c_0$  to  $c_3$ . The equivalent linear system is:

- 1)  $c_2 + c_1 = 0$
- 2)  $c_0 + c_1 = 1$
- 3)  $c_3 + c_0 = 0$
- 4)  $c_2 = 0$

Solving this linear system gives:  $c_0 = 1, c_1 = 0, c_2 = 0, c_3 = 1$ , i.e., the polynomial is  $x^4 + x + 1$ .

In a case where there is no known sequence of  $2n$  bits, but shorter sequences, if we can create a linear system of  $n$  independent equations, we can solve it to get the polynomial. In practice, it is most probable that an adversary will have enough sequences to apply this attack.

If the length of the LFSR is not known, an adversary will need a longer sequence and it would take a bit more time. He can guess the length and try to deploy this method, checking the results against the sequence that he holds, until it converges to the correct length.

## 3.2 Berlekamp-Massey Algorithm

The Berlekamp-Massey [24, 25] algorithm finds the shortest LFSR that generates an infinite binary sequence  $s$ , given a long-enough finite subsequence of  $s$ .

To understand the relation between the infinite sequence and the finite sequence, we define first the *linear complexity* of a sequence (finite or infinite):

**Definition 4** *The linear complexity of a sequence  $s$  is the length of the shortest LFSR that generates  $s$ <sup>1</sup>.*

A sequence of length  $2n$  bits is required to deploy the algorithm for a sequence of linear complexity  $n$ . This is better than the linear system attack, since here the length is unknown.

---

<sup>1</sup>If  $s$  is finite, say of length  $k$  this LFSR generates  $s$  as its first  $k$  output bits

The running time of the algorithm is  $O(n^2)$  bit operations.

Figure 3.1 depicts the algorithm. During the iterations a LFSR is built, giving us, at the end, the shortest LFSR that generates the given sequence. Figure 3.2 shows the steps of Berlekamp-Massey algorithm on a the same sequence we used in 3.1, 01100100.... The result is a LFSR of length 4 and the polynomial  $P = 1 + x + x^4$ .

*Notations and variables:*

$S^k = S_0, S_1, \dots, S_{k-1}$	a given sequence of length $k$ bits
$P = \sum c_i x^i + 1$	the LFSR polynomial
$n$	the <i>linear complexity</i> of $S^k$
$l \leq k$	a counter, the number of bits being used so far
$Q$	LFSR that generates the first $m$ bits of $S^k$ and is shorter than $l$
$m$	the <i>linear complexity</i> of $Q$
$T$	a polynomial
$d \in \{0, 1\}$	the <i>next discrepancy</i> —the difference between the next bit in the sequence and the next bit that is generated by the LFSR

*Initialization:*

$P := 1, n := 0, Q := 1, l := 0, m := -1$

*Main Algorithm:*

```

while ( $l < k$ )
   $d := S_l + \sum_{i=1}^n c_i S_{l-i} \pmod{2}$ 
  if ( $d = 1$ ) then
     $T := P$ 
     $P := P + Qx^{l-m}$ 
    if ( $n \leq l/2$ ) then
       $n := l + 1 - n$ 
       $m := l$ 
       $Q := T$ 
    end if
  end if
   $l := l + 1$ 
end while

```

**Figure 3.1:** Berlekamp-Massey Algorithm.

$S_l$	$d$	$T$	$P$	$n$	$m$	$Q$	$l$
	–	–	1	0	–1	1	0
0	0	–	1	0	–1	1	1
1	1	1	$1 + x^2$	2	1	1	2
1	1	$1 + x^2$	$1 + x + x^2$	2	1	1	3
0	0	$1 + x^2$	$1 + x + x^2$	2	1	1	4
0	1	$1 + x + x^2$	$1 + x + x^2 + x^3$	3	4	$1 + x + x^2$	5
1	0	$1 + x + x^2$	$1 + x + x^2 + x^3$	3	4	$1 + x + x^2$	6
0	1	$1 + x + x^2 + x^3$	$1 + x + x^4$	4	6	$1 + x + x^2 + x^3$	7
0	0	$1 + x + x^2 + x^3$	$1 + x + x^4$	4	6	$1 + x + x^2 + x^3$	8

**Figure 3.2:** Berlekamp-Massey Example. Steps of the Berlekamp-Massey algorithm on a given sequence 01100100.

### 3.3 Wagner and Goldberg Attack on the Netscape Browser

Wagner and Goldberg [26] revealed a flaw in the generation of random numbers in the Netscape’s implementation of SSL. Soon afterwards, Netscape fixed it in its browser’s new version.

SSL uses a large random number as a secret key, known only to the two parties. This number is generated by a PRNG, whose seed depends on the time of day, the process ID  $pid$ , and the parent process ID  $ppid$ . Figure 3.3 shows the seeding process.

```

global variable seed;
RNG_CreateContext()
    (seconds, microseconds) = time of day;
                                /* Time elapsed since 1970 */
    pid = process ID;
    ppid = parent process ID;
    a = mklcpr(microseconds);
    b = mklcpr(pid + seconds + (ppid << 12));
    seed = MD5(a, b);

```

**Figure 3.3:** The Netscape Seeding Process Pseudo Code.

The functions `mk1cpr()`<sup>2</sup> and `MD5()` are assumed to be known by an adversary, so the unpredictability of the RNG depends only on *seconds*, *microseconds*, *pid*, and *ppid*.

An attacker can guess *seconds*, using a sniffing tool, and *microseconds* is guessable by a brute-force (there are  $10^6 = 2^{20}$  microseconds in a second).

If the attacker has an account on the attacked machine, he can get *pid* and *ppid* using the `ps` utility.

If he does not have an account on that machine, he should use some tricks to guess *pid* and *ppid*. First, although each of these quantities is 15-bit, due to the shift operation, the sum `pid + (ppid << 12)`, has only 27 unknown bits, yielding a total of 47. Second, *ppid* is often 1, and if not, it is usually just a bit smaller than *pid*. This leaves *pid* and *ppid* with just a little more than 15 unknown bits. Third, *pid* is not secret information, and applications might leak it. As an example, Wagner and Goldberg suggest sending a message using the *sendmail* program to an invalid user in the attacked machine. This causes the machine to reply with a message that includes the last *pid* used. From this point it is usually not a long way to guess the Netscape's *pid*.

In summary, the Netscape RNG seed has between 20 to 47 unknown bits. It is definitely feasible to attack.

## 3.4 Gutterman and Malkhi Attack on Java Session-id Generation

Gutterman and Malkhi [27] show how one can exploit a flaw in the implementation of Java Servlet session-id to impersonate another client. In this section we partially describe their attack.

HTTP is a stateless protocol. In order to manage a session between the client and the server, a state must be maintained. The two ways to do it is by using cookies and URL rewriting, both based on session-id.

---

<sup>2</sup>The function `mk1cpr(x)` returns `((0xDEECE66D * x + 0x2BBB62DC)>> 1)`.

Although e-commerce requires a secure session, in many cases this is required only once for subscribing, and subsequent e-commerce communications are done through a normal browsing session. Here session-id is used to maintain the session.

Gutterman and Malkhi show how an adversary can guess the session-id and use it to impersonate another client.

Their case study is Tomcat, the Apache Java Servlet. It generates session-ids using either `/dev/urandom` or Java PRNG. The attack is valid to the Java PRNG case.

Java PRNG has two versions, one is `java.util.Random`, and the second is `java.security.SecureRandom`. Both generate pseudo random numbers using recursive function, starting with an initial seed. This seed is computed as a mixing function (a combination of few xor operations) on the time-of-day of the server's uptime in milliseconds and the `toString()` value of `org.apache.catalina.session.ManagerBase.java`.

Assuming the server's uptime is guessable in an accuracy of a day, the time-of-day is one of approximately  $2^{26}$  optional values. In the worst case that the adversary can only guess the uptime year, the number of possible values is about  $2^{35}$ .

The Java Object methods `toString()` returns a String whose value is `getClass().getName()+"@"+Integer.toHexString(hashCode())`, from which only the result of the method `hashCode()` is not fixed. When examining the method `hashCode()` Gutterman and Malkhi discovered that some implementations (e.g. the Microsoft Windows platform) use LCG. This makes the `hashCode()` value predictable. In practice, they show that this value contributes not more than 8 unpredictable bits.

Summing these two results yields between 32 to 43 bits of entropy, which is feasible to come over by brute-force. To prevent the server side from detecting the attack, they suggest an mostly off-line procedure. In general, the attacker obtains a legitimate session-id, and verifies the guessed seed against it. Once he reveals the seed, he can get the session-ids and use them.

# Chapter 4

## Related Work—OS-based PRNGs

In the past, PRNGs were either a separate program or a standard library within a programming language. Examples include C's `rand()` and Java's `java.util.Random` which are based on LCG, and Java's `java.security.SecureRandom` which is based on SHA-1<sup>1</sup>.

Software engineering and operating system evolution introduced PRNGs which are part of the operating system. From a cryptographic point of view, this introduction brings three main advantages. One is the ability to introduce more complex algorithms with unique kernel optimization in implementation. Second is the ability to use kernel based entropy events as input to the generators. Third is the fact that in a multiuser, multi-threaded environment, many consumers can read random bits. The latter can prevent an adversary from the ability to read a long stream of consecutive bits from the PRNG.

True random numbers generators are based on physical phenomena such as radioactive decay, thermal noise in semiconductors, sound samples taken in a noisy environment, and even digitized images of a lava lamp. [12] recommends using such generators. Recently, hardware random number generators have become common, and are integrated into chipsets. For example, Intel RNG<sup>2</sup> [28] samples thermal noise

---

<sup>1</sup>To be precise, `SecureRandom` enables the caller to request a particular RNG algorithm. Sun's default implementation uses SHA-1.

<sup>2</sup>One of the Intel 810 chipset components is the 82802 firmware hub that contains a hardware RNG.

in resistors and, after some internal processing, feeds it periodically into a SHA-1 based PRNG.

In this thesis we are dealing with LRNG, which is an operating-system PRNG, therefore, we survey other operating-system PRNGs: FreeBSD [29] and Castejon-Amenedo et al. [30].

In general, both are pretty similar to LRNG. The high level design includes entropy sources, pools, and output. The entropy sources are the same that are used in LRNG— keyboard, mouse, network devices, disk, and operating system interrupts. The systems differ in the pool sizes, their mixing mechanisms, and in the output generating process. A major difference is that none of them use a feedback loop, i.e., consuming entropy is not used to modify the inner state of the entropy pools.

FreeBSD operating system PRNG is presented in [29]. Events' time stamps are used as entropy source and later hashed using AES [31] into two pools, each of 256 bits. When output is extracted, AES encryption is used as a hash function. It repeatedly encrypts a 256-bit counter with an encryption key that is taken from the entropy pools. The PRNG reseeding is simply replacing this key.

To estimate the incoming entropy, the minimum of three factors is taken: a constant per adding-entropy function-call; 1/2 of the number of the transferred bits; and per-source statistically determined measure. The first two are upper limits, and the third is not implemented. The FreeBSD implements a single non-blocking device and the authors declare their preference of performance over security.

Castejon-Amenedo et al. [30] proposes a PRNG for UNIX environments. Their system is composed of an entropy daemon and a buffer manager, that handles two devices—blocking and non-blocking. The buffer manager divides the entropy equally between the two devices, so that there is no entropy that is used in both.

The *blocking device* holds a cyclic buffer, with two pointers—one for reading and one for writing. As long as the buffer is not full, writing is possible, and as long it is not empty—reading is possible. Reading and writing are synchronized such that there are no duplicate readings of the same data and no overrides occur. Before the data is output, it is hashed using AES (keyed by data taken from the entropy sources), and cached if necessary. The unit size for reading and writing is 1 byte. The *non-blocking*

*device* holds a buffer which is organized in 16-bytes blocks. Reading is done in blocks units, and is hashed and cached as in the blocking device. Writing is done in bytes units, but an in-block fairness and inter-blocks fairness are kept. They mention a rate-attack (more reads than writes) and note that due to the AES hashing, breaking their scheme is as unfeasible as breaking AES.

A notable advantage of this scheme is the absolute separation between blocking and non-blocking devices. This avoids a possibility of denial-of-service attacks on the blocking device by using the non-blocking device, such as the ones we later present in Chapter 6.

Figure 4.1 summarizes a comparison of OS-based PRNGs. Fundamental and notable differences are in the input mechanisms, entropy estimation methods and the existence of a feedback operation. LRNG is described in detail in the following Chapter 5.

	<i>FreeBSD</i>	<i>Castejon – Amenedo</i>	<i>LRNG</i>
<i>Entropy Sources</i>	keyboard, mouse, disk, interrupts	keyboard, mouse, disk, interrupts	keyboard, mouse, disk, interrupts
<i>Input Mechanism</i>	hash using AES	divide between the two devices	TGFSR variant (see Section 5.3)
<i>Output OWF</i>	AES	AES	SHA-1
<i>Entropy Estimation</i>	$\max(\text{constant}, \text{input bits}/2)$	varries according to number of bits in the pool	based on events timings (see Section 5.4)
<i>Pool's Sizes in Bits</i>	$2 \times 256$	not specified	$4096 + 1024 + 1024$
<i>Feedback Operation</i>	no	no	yes
<i>Non – Blocking Device</i>	yes	yes	yes
<i>Blocking Device</i>	no	yes	yes
<i>Both Devices Use the Same Input</i>	not relevant	no	yes

**Figure 4.1:** Comparison of OS-Based PRNGs.

## Chapter 5

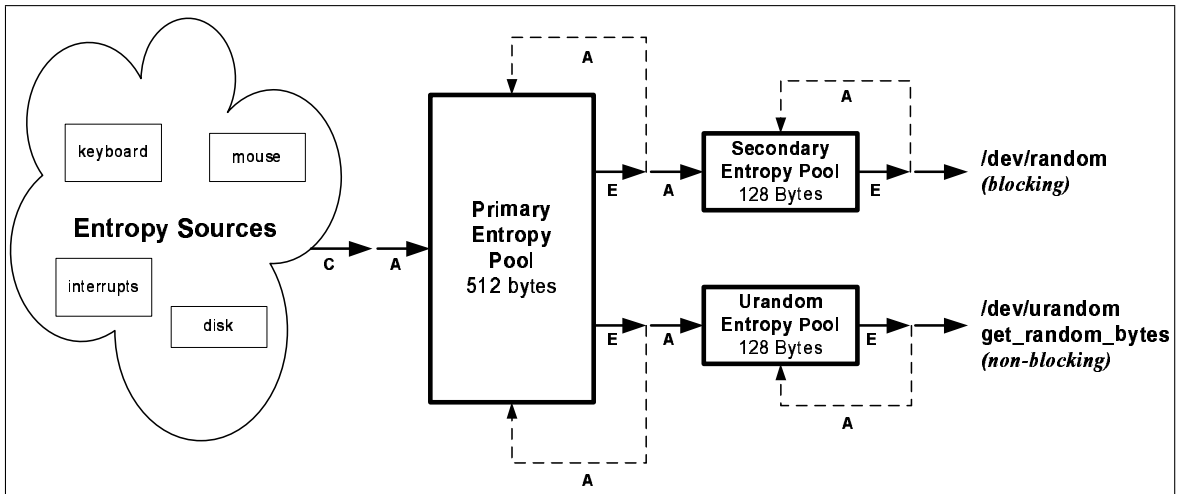
# Linux Random Number Generator Structure

The Linux random number generation can be described as three consecutive asynchronous procedures. First, operating system entropy is collected from the various events inside the kernel. In the second stage, entropy is fed into a TGFSR-like pool, using a mixing function. When random bits are requested, the last stage occurs and output is generated.

Figure 5.1 describes the LRNG flow. The internal state is kept in three entropy pools: *primary*, *secondary* and *urandom*, whose sizes are 512, 128 and 128 bytes, respectively. Entropy sources add data to the primary pool, while LRNG output is extracted from the secondary pool and from urandom pool. During the extraction operation, the inner state of the pool is modified in a feedback manner. When necessary, entropy is transferred from the primary pool to the other two.

Each pool holds its own entropy estimation counter. This is an integer value between zero and the pool size in bits (4096 for the primary, 1024 for the others), which indicates the number of bits that are considered random currently in the pool. When entropy is extracted from the pool, this counter is decremented and when entropy is added, the counter is incremented.

Decrementing the entropy counter is always by the number of extracted bits. Incrementing is more complex. If the added bits are originating from one of the



**Figure 5.1:** General Scheme of LRNG. Entropy is being collected (C) from four sources and is added (A) to the primary pool. Entropy is extracted (E) from the secondary pool or from urandom pool. Whenever entropy is extracted from a pool, some of it also feedbacks this pool (broken line). The secondary pool and the urandom pool draw in entropy from the primary pool.

entropy sources, than their entropy is estimated, and the counter is incremented accordingly. The entropy estimation uses the timings of the last few events of the same entropy source. If the entropy bits are transferred from another pool, the entropy counter is incremented by the number of transferred bits.

The entropy counter plays an essential role when extracting entropy using the blocking interface, `/dev/random`. Its task is to determine whether there is enough entropy in the secondary pool to supply the requested amount of random data. If the answer is negative, the LRNG tries to transfer entropy from the primary pool, and if this fails, it blocks and waits until some entropy input arrives and increments the entropy counter.

Entropy bits are originating from one of four sources: mouse and keyboard activity, disk I/O operations, and specific interrupts. When such an event occurs, it produces 32 bits representing its timing and 32 bits encoding its attributes (e.g., which key was pressed). These 64 bits are batched in order to be added later to the entropy pools.

In addition, the differences between the timings of successive events of the same type are used to evaluate the entropy extent of this event. In Section 5.4 we define

this procedure in details.

A few times a minute, the batched entropy data is flushed to the pools. Normally it is flushed to the primary pool and if it is full, it uses the secondary. When the secondary pool is full it moves back to the primary, and so on. It is never flushed to the urandom pool. This flushing adds entropy to the pool and increments its entropy counter according to the estimation that was calculated when the entropy was batched.

The entropy-addition is not simply copying bits from the batch storage to the pools. It uses a mixing algorithm, based on TGFSR. For this purpose, each pool holds a primitive polynomial. The polynomial is chosen according to the pool's size, so the secondary and the urandom pools have the same polynomial.

The last part in the puzzle is consuming random bits. This may be done by three interfaces. Random bits are extracted from one of the three pools: from the urandom pool when the user uses `/dev/urandom` and when the kernel calls `get_random_bytes`; from the secondary pool when the user uses `/dev/random`; and from the primary pool when one of the two others does not have enough entropy and needs re-filling. The process of entropy extraction includes three steps. One is the actual extraction—random bits are supplied to the consumer. Second is decrementing the entropy counter of the pool. Third is modifying the pool's content. This process involves hashing the pool using SHA-1, and adding intermediate hashing results to this pool. This addition is done in the same way that out-sourced entropy is added, except for not incrementing the entropy counter.

We study each of the LRNG steps in the following sections, starting with the pool initialization in Section 5.1 and the various entropy sources in Section 5.2, followed by a detailed description of the algebraic process of adding the entropy to the pool in Section 5.3. Section 5.4 studies how the amount of entropy inside the LRNG pools is measured. Extraction of random data is described in Section 5.5.

## 5.1 Initialization

Operating system start-up includes a sequence of routine actions. This sequence, including initializing the LRNG with constant OS parameters and time-of-day, can easily be predicted by an adversary. If no special actions are taken, the LRNG state will include very low entropy<sup>1</sup>.

To solve this problem, LRNG simulate continuity along shut-downs and start-ups. This is done by “skipping” system boots. A random-seed is saved at shut-down and is written to the pools at start-up. A script that is activated during system start-ups and shut-downs uses the read and write capabilities of `/dev/urandom` interface to perform this maintenance.

The script saves 512 bytes of randomness between boots in a file. During shut-down it reads 512 bytes from `/dev/urandom` and writes them to the file, and during start-up these bits are written back to `/dev/urandom`. Writing to `/dev/urandom` modifies the *primary* pool and not the `urandom` pool, as one could expect. The secondary and the `urandom` pool get their entropy from the primary pool, so the script operation actually affects all three pools.

Since the random-seed is saved in a file on the hard disk, anyone who can physically access the disk, can read that file and know the seed. Permission limitation is not too helpful here, since it is related to the operating system, not to the hard disk.

It is important to note, that this script is part of a distribution package such as RedHat and not part of the kernel source code itself. The author of [32] instructs Linux distribution developers to add this script in order to ensure unpredictability at system start-ups. This implies that the security of the LRNG is not completely stand-alone but dependent on an external component which can be predictable in a certain Linux distribution<sup>2</sup>.

---

<sup>1</sup>The time of day is given as seconds and micro-seconds, each is 32-bits. In reality this has very limited entropy as one can find computer uptime within an accuracy of a minute, which leads to a brute-force search of  $60_{seconds} \times 10^6_{microseconds} < 2^{26}$  which is feasible. Even if the attacker cannot get the system uptime, he can check the last modification time of files that are created or modified during the system start-up, and know the uptime in an accuracy of minutes.

<sup>2</sup> An extreme case are Linux systems that are booted from a CD, e.g., Knoppix. These distributions cannot obey that instruction simply because they do not have where to save the random-seed between shut-down and start-up. As a result they lack this initialization phase

## 5.2 Collecting Entropy

LRNG collects entropy from events originating from the keyboard, mouse, disk and interrupts. When such an event occurs, two 32-bit words are used as input to the entropy pools: the timing of the event (in **jiffies**<sup>3</sup> granularity or in cpu-cycles granularity<sup>4</sup>), and its **type-value** defined as follows:

- *keyboard event.*

Keyboard press and release codes, valid range between [0, 255]. For example, Esc press code is 01, and its release code is 81. The number of unknown bits in this value is 8.

- *mouse event.*

Computed using

$$\text{type-value} := (\text{type} \ll 4) \oplus \text{code} \oplus (\text{code} \gg 4) \oplus \text{value}$$

where *type* describes the event type - pressing or releasing in case of mouse buttons and start movement or end movement in case of mouse movement; *code* is the mouse button pressed (left, right or middle) or wheel scrolling in case of mouse buttons, and which axis the movement is (horizontally or vertically) in case of mouse movement; *value* is true/false in case of mouse buttons press or release<sup>5</sup>, 1 or -1 for denoting scrolling direction (1 for up, -1 for down) in case of wheel scrolling, and the size of movement in case of mouse movement. In short, the mouse data is a 32-bit word with the movement size as its main entropy factor. However, only 10 bits are used for movement, another 2 bits are used for the buttons, so in fact only 12 out of the 32 bits are effective.

- *disk event.*

Computed at completion of a disk (such as IDE, SCSI, Floppy, block devices)

---

<sup>3</sup>*jiffies* is the number of clock-ticks (each is 10 milliseconds) from the time the machine was booted.

<sup>4</sup>Currently cpu-cycles granularity is only used on SMP.

<sup>5</sup>Each action produces an input for all three buttons to the mouse type-value formula. The button that was active gets an action *value* = 1 while the others get *value* = 0.

I/O operation. Its type-value is composed of *major* and *minor* numbers<sup>6</sup> as:

$$\text{type-value} := 0x100 + ((\text{major} \ll 20) | \text{minor})$$

If there is only one IDE disk, the type-value is fixed, since the major and minor numbers are constants<sup>7</sup>. Assuming an average machine has no more than 8 disks, the type-value actual span is limited to 3 bits.

- *interrupt event*.

The result of an interrupt occurrence is the IRQ (interrupt request channel) number, with a valid range of [0,15]. It is important to note that as of the current kernel versions, only very limited number of hardware device drivers supply interrupt addition into the LRNG. In many setups interrupts will not add any entropy events.

## 5.3 Adding Entropy

Entropy is added to the pools in three conditions: collecting entropy from the sources of random data and flushing it to the primary or secondary pool; transferring entropy from the primary pool to the secondary or urandom pool; writing to `/dev/{u}random`.

Normally, adding entropy is to the primary pool. There are three exceptions: self feedback when extracting entropy; transferring entropy from the primary to the one of the other pools; flushing to the secondary pool when the primary pool is full (contains maximum entropy).

The mechanism of adding entropy to a pool is based on TGFSR (see Section 2.3.4). For this purpose, each pool maintains a primitive polynomial. The primary pool's polynomial is  $x^{128} + x^{103} + x^{76} + x^{51} + x^{25} + x + 1$ . The secondary and the urandom pool have the same polynomial:  $x^{32} + x^{26} + x^{20} + x^{14} + x^7 + x + 1$ .

The TGFSR matrix multiplication is implemented by xoring with a value from a

<sup>6</sup>*major* and *minor* numbers are operating system's symbols that together uniquely define a device.

<sup>7</sup>In most cases the major is 3 (first IDE disk) and the minor is 0 (master), and their combined type-value yields 0x300100.

twist-table, whose values are: `0`, `0x3b6e20c8`, `0x76dc4190`, `0x4db26158`, `0xedb88320`, `0xd6d6a3e8`, `0x9b64c2b0`, `0xa00ae278`.

In all FSR variants, including TGFSR, the only input is the initial seed. In each iteration the internal state is used to generate the next output and update the state.

LRNG entropy-addition differs from TGFSR by adding entropy on each iteration while TGFSR is only seeded once. In TGFSR, the resulted output and the inner state are functions of this initial seed. In LRNG entropy is added in each round of state and output computation. The following equation defines the LRNG entropy-addition. Note that  $g$  is the new entropy element

$$a_i = g \oplus a_{i-p+q} \oplus a_{i-p}A \quad ; \quad i = p, p+1, \dots$$

The entropy addition can be analyzed in the two following ways. The first is to have a reseeding process in each iteration and the second is using the TGFSR to encrypt the entropy input.

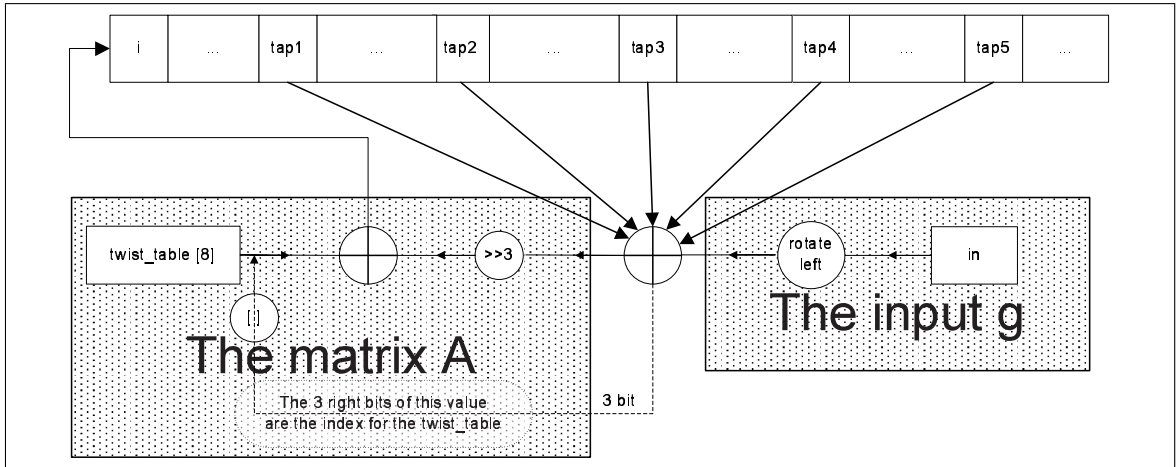
LRNG reseeding process changes the elementary properties of the TGFSR. We can no longer guarantee the long period, nor that it is linear. Hence, Berlekamp-Massey [24, 25] prediction algorithm for LFSR (see Section 3.2) is not applicable in this case. However, this also does not prove that the LRNG framework is secure.

Figure 5.2 depicts the LRNG entropy-addition in details.

## 5.4 Estimating the Entropy Amount

We described earlier the entropy sources of LRNG and the way entropy is added to the pools. One of the fundamental issues is estimating the amount of entropy which is added to the pool in each occasion, and more generally, estimating the current amount of entropy in the pools.

This is the essence of the difference between `/dev/random` and `/dev/urandom`. The first interface (`/dev/random`) does not return more bits than the entropy estimation and thus might block. The second interface (`/dev/urandom`, and so does the kernel interface—`get_random_bytes`) returns any number of pseudo-random bits, according



**Figure 5.2:** LRNG FSR. There are three main components in LRNG FSR: the pool, the input and the matrix, represented by a twist-table (an array of 8 “magic” numbers). Entropy is added to a pool in word (32 bits) units, for each word the following process occurs. (1) Five words (the “taps”) and the  $i^{\text{th}}$  word of the current content of the pool are xored. (2) The input  $g$  is rotated left by a quantity that is incremented modulo 31 by 7 (or by 14, when  $i = 0$ ). This is xored with the result of step 1. (3) The 3 least significant bits of the result of step 2 determines the index of the `twist_table` that in turns is xored with the other 29 bits. The outcome of step 3 is written to the  $i^{\text{th}}$  word in the pool.

to the request. This difference implies that the entropy estimation is important only for the more secure interface—`/dev/random`.

As described in Section 5.2 the input entropy is the events’ timings and their type-values. LRNG estimates the entropy addition as a function of the timings only, in the following manner:

**Definition 5** Let  $e_n$  denote event number  $n$ , and  $t_n$  denote its timing. We define

$$\begin{aligned}\delta_n &= t_n - t_{n-1} \\ \delta_n^2 &= \delta_n - \delta_{n-1} \\ \delta_n^3 &= \delta_n^2 - \delta_{n-1}^2\end{aligned}$$

Note that  $t_n$ ,  $\delta_n$ ,  $\delta_n^2$ ,  $\delta_n^3$  are each 32 bits long.

The amount of entropy added by  $e_n$  is:  $\log_2 (\min (|\delta_n|, |\delta_n^2|, |\delta_n^3|)_{[19-30]})$  where  $S_{[a-b]}$  denotes bits  $a$  to  $b$  (inclusive) of  $S$  ( $0 \leq a \leq b < S$ 's length and 0 is the MSB).

This estimation is relevant only in the case of adding entropy from the entropy sources to the pools. In other cases, the entropy estimation is simpler. When a user writes to one of the device drivers `/dev/{u}random`, the entropy estimation is not incremented, since the system cannot estimate the entropy of the user input. When extracting  $m$  bytes from a pool, the entropy estimation is decremented by  $8m$  bits. When entropy data is transferred from one pool to another, the first pool estimation is decremented, and the second's is incremented—both by the amount of transferred bits.

## 5.5 Extracting Random Bits

Entropy is extracted from the secondary pool in case of `/dev/random` and from the urandom pool in case of `/dev/urandom` or `get_random_bytes`. If there is not enough entropy in the pool (secondary or urandom, respectively), it is refilled from the primary pool. The refilling operation itself is composed of extracting entropy from the primary pool, and adding it to the refilled pool. Entropy estimations of the participating pools are updated according to the transferred entropy amount.

Extracting entropy from a pool is not a simple operation. It involves hashing the extracted bits, modifying the pool inner-state and decrementing the entropy amount estimation by the number of extracted bits.

Figure 5.3 presents a pseudo code of the extracting algorithm. For simplicity it does not include the entropy estimation decrement and the refilling process. The algorithm describes extracting random bytes from the secondary or urandom pool, which is the common case<sup>8</sup>. Extracting entropy is done by units of up to 10 bytes. It uses SHA-1 and entropy-addition operations before actually outputting entropy in order to avoid backtracking attacks. In addition it uses folding to blur recognizable patterns.

---

<sup>8</sup>Extracting from the primary pool is performed during refilling one of the other pools. In this case, the algorithm is slightly different. In general, the SHA-1 and the `add` operations are performed  $x + 1$  times, when  $x$  is computed as the pool size (in 32-bit words) divided by 16. In Figure 5.3 we refer to the secondary or urandom pool, whose both sizes are 32 words, so these operations are performed 3 times. For the primary pool, whose size is 128 words, these operations are performed 9 times.

Note that only the first SHA-1 in each iteration uses SHA-1's 5 initial constants values<sup>9</sup>. Successive SHA-1 operations use the 5 words of the previous SHA-1 result instead. The first SHA-1 uses the first half of the pool, the second SHA-1 uses the other half, and the third uses half of the pool, starting from the last modified word, and backwards. In each iteration during the extracting operation, three words are added to the pool. These additions modify three words of the pool, in order to make backtracking attacks more difficult, but are not considered as adding real entropy.

```

Algorithm Extract(pool, nbytes, j):
  while nbytes > 0
    tmp := SHA-1(pool[0..15])
    pool[j] := add(tmp[0])

    tmp := SHA-1(pool[16..31])
    pool[(j-1) mod 32] := add(tmp[2])
    pool[(j-2) mod 32] := add(tmp[4])

    tmp := SHA-1(pool[(j-2) mod 32..(j-2-15) mod 32])
    tmp := folding(tmp[0..4])
    output(tmp, min(nbytes, 10))

    nbytes := nbytes - min(nbytes, 10)
    j := (j-3) mod 32
  end while

```

**Figure 5.3:** Extracting Algorithm Pseudo Code. *pool* is the pool to extract entropy from, *nbytes* is the number of requested bytes, and *j* is the current position in *pool* to be modified when entropy is added.

After this process, the 5 left words (20 bytes) are folded as described in Figure 5.4. The first word is xored with the fourth word (written to the first word), the second word is xored with the fifth (written to the second word), and the third word left half is xored with its right half and the result is written to the left half.

At the end of each iteration, up to 10 bytes from the left side of the folded result are copied to the target (user or kernel) buffer, and the number of bytes to be copied is updated. The loop continues until the requested number of bytes are output.

<sup>9</sup>See [33] for SHA-1 definition.

$$\begin{aligned} \textit{input} & : W_0, W_1, W_2, W_3, W_4 \\ \textit{output} & : W_0 \oplus W_3, W_1 \oplus W_4, W_{2_{[0-15]}} \oplus W_{2_{[16-31]}} \end{aligned}$$

**Figure 5.4:** Folding Operation. Folding 5 words (160 bits) to 2.5 words (80 bits).  $W_i$  denotes the  $i^{\text{th}}$  word,  $W_{i_{[l-m]}}$  denotes bits  $l - m$  (inclusive) of the  $i^{\text{th}}$  word. The folding uses xor operation on words and half-words.

# Chapter 6

## Attacks, Weaknesses and Security Flaws

### 6.1 Denial of Service

There is no limitation on the number of bits a user can read from the random devices per time unit. However, the secure interface `/dev/random` blocks user reading once there is not enough entropy within the kernel until additional “noise” is added to the pools. These two facts together present two denial of service attacks which will block all users from reading `/dev/random` bits.

The first attack is simply to read bits from `/dev/random`. As there is no limit and no prioritization, this will result in blocking other users and an adversary may use it to delay them for a long period of time.

Furthermore, this attack can even be done remotely with system requests for random bytes (`get_random_bytes`) in a significantly higher rate than the entropy input events. Since the non-blocking pool is refilled from the blocking pool, it will result with the same denial-of-service result. A simple way for an adversary to issue this attack may be to set many TCP connections. For each connection, a TCP-syn-cookie is generated, which requires 128 bytes from the non-blocking pool, hence reducing the entropy count.

**Solution.** As entropy is a limited and essential resource in LRNG and its consumption must be controlled, operating systems' common solution for such an event is through definition of a new quota per user or group for the consumption of random bits.

## 6.2 Predictability of `/dev/urandom`

If there are only calls to `/dev/urandom` and `/dev/random` and no calls to `get_random_bytes`, the `urandom` pool does not get any external input and its state is a function of its initial state and the self feedbacks only. The lack of input is composed of two factors: First, entropy originating from the sources is added to the primary pool only, and not to the `urandom` pool. Second, in contradiction to `get_random_bytes`, when `/dev/urandom` is used, the `urandom` pool is not refilled from the primary pool.

**Solution.** A simple solution to this defect is making `/dev/urandom` equal to `get_random_bytes` in the sense that the `urandom` pool is refilled from the primary pool whenever necessary. It is important to note that while this solves the predictability issue, it introduces the denial-of-service attack which must be solved according to the guidelines above.

## 6.3 Uninitialized Memory Read

The entropy extraction from the primary into the secondary pool (see Section 5.5) includes usage of uninitialized memory, where instead of adding  $n$  bytes to the secondary pool,  $4 \times n$  bytes are added. This entropy transfer between pools is performed by the function `xfer_secondary_pool` in the code presented in Figure 6.1<sup>1</sup>:

The problem is that extracting is by units of bytes and adding is by units of words (one word is four bytes). So actually, the amount of “entropy” that is added to the

---

<sup>1</sup>The parameters meanings are: `random_state`—the primary pool, `tmp`—a buffer that stores the extracted entropy, `bytes`—the number of bytes to extract, `EXTRACT_ENTROPY_LIMIT`—a flag that denotes that it is a blocking extraction, `r`—the secondary pool (or the `urandom` pool).

```
1335         bytes=extract_entropy(random_state, tmp, bytes,
1336                               EXTRACT_ENTROPY_LIMIT);
1337         add_entropy_words(r, tmp, bytes);
1338         credit_entropy_store(r, bytes*8);
```

**Figure 6.1:** Transferring Entropy from the Primary Pool (`random_state`) to the Secondary (or `urandom`) Pool (`r`). A code segment from `xfer_secondary_pool`.

secondary pool is 4 times larger than the amount that is extracted from the primary pool.

The result is that only one quarter of the entropy that is added to the secondary pool is taken from the primary pool, and the remaining three quarters are taken from an arbitrary memory. In addition to being a programming bug, this is a security hole. If someone can access that memory, he or she can control, or at least know, three quarters of the secondary pool entropy input, and gain control or knowledge of its inner state. A side effect that is not depended on the existence of an adversary is a mistake in the entropy estimation because the arbitrary memory is not necessarily random and might even stay unchanged upon many calls to `xfer_secondary_pool`.

Another effect might be a segmentation fault, if the function `add_entropy_words` tries to read from a memory area that does not exist.

**Solution.** This is a simple programming bug. Instead of adding `bytes`, add `bytes/4`.

## 6.4 Guessable Passwords

Usually, the first user-operation in a computer system is logging in, and its first input is the password, or the user-name and password.

In a scenario of a disk-less system, without a random-seed that is saved between start-ups, we can imagine a situation where an attacker can know the initial state of the LRNG. Since the code is open source, he or she can also imitate the LRNG operation. If in addition, he can read random bits fast enough, he is able to identify whether the LRNG has input between two successive reads.

Recall that we refer to disk-less system, and assuming the system uses network

devices whose interrupts do not yield entropy<sup>2</sup>, the first input is from the keyboard. The possible space for this input's entropy is the number of possible type-values times the number of possible timings. The first factor is 256, i.e.,  $2^8$ , and the second depends on the reading speed. If the attacker reads every second, it is 100 jiffies, i.e. approximately,  $2^7$ .

In total, finding the first letter costs  $2^{15}$  guesses. The same goes for all the user-name and password characters, yielding a running time of  $2^{15} \times n$  when  $n$  is the number of characters. This is efficient attack in comparison to a simple brute-force, that costs  $2^{15n}$ .

**Solution.** Remove the keyboard keys involvement in LRNG. Keyboard entropy should be only its timings, and not its type-values. This will prevent guessing the password, and will not affect the entropy estimation.

## 6.5 Creating Noise that Directly Affects the LRNG Output

Normally, there is a separation between input and output of LRNG, but when the primary pool is full, the batched entropy is flashed directly to the secondary pool, from which it is output when using `/dev/random`. The direct flow between the input and the output sides of LRNG gives an adversary the ability to create noise that directly affects the output.

**Solution.** Always flush the batched entropy to the primary pool, even if it is full. This makes a strict separation between the input and the output sides of LRNG.

---

<sup>2</sup>e.g., 3Com PCI 3c905B Cyclone 100baseTx.

# Chapter 7

## Discussion and Conclusions

This thesis presents an analysis of Linux PRNG, an entropy based intra operating system PRNG. Our analysis presents the complex structure of the LRNG from its entropy collection procedure through mixing and to the extraction function based on one-way functions.

In Chapters 2, 3 and 4 we bring some backgrounds about PRNGs, known attacks on PRNGs and other operating-systems PRNGs. In Chapters 5 and 6 we present and analyze the Linux PRNG.

In summary, our impression is that LRNG is a strong cryptography mechanism for producing random numbers. Its main entropy source is the timings of the disk I/O operations, which are affected from truly random variables such as air turbulence and the temperature of the mechanical parts of the disk drive. These are proved experimentally to produce about 100 truly random bits a minute ([12]). We found that the disk operation timing is so sensitive that we could not get the same timings even of the first disk operation that affects LRNG (the minimal difference was about  $2^{16}$  jiffies).

In this work we did not examine the statistical aspect of LRNG. This was studied by Nylund and Runds [34]. They found that LRNG passed well both the Diehard and the Nist tests.

We did not find an attack which enables a realistic prediction result that can compromise the entire cryptographic protocols based on the LRNG. Nevertheless,

our results prove that there are use cases where the LRNG is predictable or includes security holes such as denial of service attacks.

We began our research on Linux kernel 2.4.25, moving to versions 2.6.*x*. Some of the versions include significant changes, e.g., adding the urandom pool (first introduced in version 2.6.9).

Our study was conducted on the latest Linux kernel at the time, labeled version 2.6.10 which was released on December 24, 2004. Since then, the kernel kept developing. Version 2.6.11 was released on March 2005 and patches are being published since then<sup>1</sup>.

In version 2.6.11 the usage of uninitialized memory we described in Section 6.3 was fixed, while the rest of the security holes remained the same. Since then, the rest of the patches, numbered 2.6.11.*x*, and 2.6.12.*x*, use the simple solution we described in Section 6.2 to solve the `/dev/urandom/` predictability problem. Also, a special mechanism is maintained to avoid denial-of-service attacks—`/dev/urandom` and `get_random_bytes` requests cannot empty the primary pool, at least 16 bytes are reserved. Indeed denial-of-service attack is harder, but still possible, since after many `/dev/urandom` and `get_random_bytes` requests, a single `/dev/random` request of 16 bytes empties the primary pool. Another change is the stricter separation between input and output—the primary pool plays solely the input side, and the other two are the output side. In addition, the entropy-batching is removed. This makes the last flaw (Section 6.5) irrelevant.

These patches from the last months also introduce new security holes.

- In the extraction process, the input of the second hash is 16 words from the pool, starting from the 17<sup>th</sup> byte, instead of the 17<sup>th</sup> word. This bug is caused due to a wrong casting to 8 bits data-type.
- In the data structure that represents a pool, `entropy_store`, there is a pointer to the pool from which this pool may be refilled (`pull`), and a flag to signal whether extraction from this pool is limited (`limit`). The first field (`pull`) is

---

<sup>1</sup>See <http://www.linuxhq.com/kernel/file/drivers/char/random.c> for the `random.c` incremental changes.

---

not initialized in the primary pool, and the second (`limit`) is not initialized in the urandom pool. These fields get unknown values and may cause undefined behavior.

During the writing of this document, version 2.6.12 was released. In this version, alongside many improvements, we found a apparent bug in the last part of the folding operation implementation. Instead of xoring the two halves of the middle (third) word and writing the result to it left half (see Section 5.5 and Figure 5.4), in version 2.6.12 the fourth word is being cyclic shifted by 16 bits (i.e., its two halves change), xored with the first word and the result is written to the first word. The middle word is left untouched.

```
buf[0] ^= buf[3];
buf[1] ^= buf[4];
buf[0] ^= rol32(buf[3], 16);
```

We would expect the last line to be `buf[2] ^= rol32(buf[2], 16);`.

As far as we could tell the different Linux distributions (e.g., Redhat, Debian, Slakware) have little if any effect on the LRNG since all distributions use the same kernel source. Changes occur only within the system up and down times which to our finding are only cosmetics. As there are hundreds of different distributions it may not be true for all of them.

Our study does not cover all Linux kernel options. For example, we did not take into account multi-cpus hardware configurations, nor any unique hardware configuration such as Qtronix<sup>2</sup> unique keyboard and mouse device which has different entropy collecting formula than the one described here.

As the LRNG is a kernel based code, its reverse engineering is not simple and requires both dynamic and static analysis. To simplify the process we have developed a user mode simulator of the LRNG which can be used for further research and simulation of the LRNG. The entire application and source can be downloaded from the author's web page at <http://www.cs.huji.ac.il/~reinman>.

The LRNG is an open source project which enables an adversary not only to

---

<sup>2</sup><http://lxr.linux.no/source/drivers/char/qtronix.c?v=2.6.10>

read the entire source code but also to be able to trace changes inside the source configuration managements system. The latter enabled us to trace back and see that the security hole we presented in Section 6.3 was actually there for 17 months and was fixed just a few days after we noticed it.

Open source also has its benefits as to security. One is the ability to change the LRNG code. A cryptographer can very easily add tailored hardening to the current code which is much harder, if at all possible, for closed source PRNGs. Public review is clearly a benefit, having the code exposed to many people and allowing them to comment.

# Bibliography

- [1] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4), October 1949.
- [2] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, Internet Engineering Task Force, January 1999.
- [3] P. Zimmermann. *PGP User's Guide*. MIT Press, 1994.
- [4] M. Elkins. MIME security with Pretty Good Privacy (PGP). RFC 2015, Internet Engineering Task Force, October 1996.
- [5] J. Callas, L. Donnerhackle, H. Finney, and R. Thayer. OpenPGP message format. RFC 2440, Internet Engineering Task Force, November 1998.
- [6] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.
- [7] W. Diffie and M. Helman. New directions in cryptography. *IEEE Transactions on Information Society*, 22(6):644–654, november 1976.
- [8] B. Schneier. *Secrets & Lies — Digital Security in a Networked World*. John Wiley & Sons, 2000.
- [9] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 2001.
- [10] S. A. Teukolsky, B. P. Flannery, W. H. Press, and W. T. Vetterling. *Numerical recipes in C*. Cambridge, 1992.

- 
- [11] J. A. Reeds. ‘Cracking’ a random number generator. *Cryptologia*, 1(1), 1977.
- [12] D. Eastlake 3rd, S. D. Crocker, and J. Schiller. Randomness recommendations for security. RFC 1750, Internet Engineering Task Force, December 1994.
- [13] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proc. 2nd Sympos. on Large-Scale Digital Calculating Machinery, Cambridge, MA, 1949*, pages 141–146, Cambridge, MA, 1951. Harvard University Press.
- [14] J. von Neumann. Various techniques for use in connection with random digits. In *von Neumann’s Collected Works*, volume 5, pages 768–770. Pergamon, 1963.
- [15] T. G. Lewis and W. H. Payne. Generalized feedback shift register pseudorandom number algorithm. *Journal of the ACM*, 20(3):456–468, July 1973.
- [16] A. Shamir. On the generation of cryptographically strong pseudo-random sequences. In *Proc. ICALP*, pages 544–550. Springer, 1981.
- [17] L. Blum, M. Blum, and M. Shub. Comparison of two pseudo-random number generators. pages 61–78, New York, 1983. Plenum Press.
- [18] M. Matsumoto and Y. Kurita. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation*, 2(3):179–194, 1992.
- [19] M. Matsumoto and Y. Kurita. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation*, 4(3):254–266, 1994.
- [20] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998. <http://www.math.keio.ac.jp/matsumoto/emt.html>.
- [21] B. Schneier. *Applied Cryptography — Protocols, Algorithms and Source Code in C*. John Wiley & Sons, second edition, 1996.
- [22] J. B. Plumstead. Inferring a sequence produced by a linear congruence. In *FOCS 23*, pages 153–159, 1982.

- 
- [23] J. B. Plumstead. Inferring a sequence produced by a linear congruence. In *CRYPTO*, pages 317–319, 1982.
- [24] E. R. Berlekamp. *Algebraic Coding Theory*. McGraw Hill, 1968.
- [25] J. L. Massey. Shift register synthesis and BCH decoding. IT-15:18–27, 1969.
- [26] I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr Dobb's*, pages 66–70, January 1996.
- [27] Z. Gutterman and D. Malkhi. Hold your sessions: An attack on Java session-id generation. In Alfred Menezes, editor, *CT-RSA*, volume 3376 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2005.
- [28] B. Jun and P. Kocher. The intel random number generator. Technical report, Cryptography Research, Inc. White Paper Prepared for Intel Corporation, 1999.
- [29] M. R. V. Murray. An implementation of the Yarrow PRNG for FreeBSD. In Samuel J. Leffler, editor, *BSDCon*, pages 47–53. USENIX, 2002.
- [30] J. Castejon-Amenedo, R. McCue, and B. H. Simov. Extracting randomness from external interrupts. In *The IASTED International Conference on Communication, Network, and Information Security*, pages 141–146, USA, December 2003.
- [31] National Institute of Standards and Technology (NIST). Advanced Encryption Standard. Available on: <http://csrc.nist.gov/CryptoToolkit/aes/>.
- [32] T. Ts'o. `random.c`—Linux kernel random number generator. <http://www.kernel.org>.
- [33] D. Eastlake 3rd and P. Jones. US secure hash algorithm 1 (SHA1). Request for Comments 3174, Internet Engineering Task Force, September 2001.
- [34] T. Nylund and R. Runds. Implementing a random device driver under solaris 8. Master's thesis, Department of Computer Science, University of Gothenburg, Sweden, 2002.