Analysis of the Linux Random Number Generator

Zvi Gutterman
Safend and The Hebrew University of Jerusalem

Benny Pinkas
University of Haifa

Tzachy Reinman
The Hebrew University of Jerusalem

Abstract

Linux is the most popular open source project. The Linux random number generator is part of the kernel of all Linux distributions and is based on generating randomness from entropy of operating system events. The output of this generator is used for almost every security protocol, including TLS/SSL key generation, choosing TCP sequence numbers, and file system and email encryption. Although the generator is part of an open source project, its source code (about 2500 *lines of code) is poorly documented, and patched with hundreds of code patches.*

We used dynamic and static reverse engineering to learn the operation of this generator. This paper presents a description of the underlying algorithms and exposes several security vulnerabilities. In particular, we show an attack on the forward security of the generator which enables an adversary who exposes the state of the generator to compute previous states and outputs. In addition we present a few cryptographic flaws in the design of the generator, as well as measurements of the actual entropy collected by it, and a critical analysis of the use of the generator in Linux distributions on diskless devices.

1 Introduction

Randomness is a crucial resource for cryptography, and random number generators are therefore critical building blocks of almost all cryptographic systems. The security analysis of almost any system assumes a source of random bits, whose output can be used, for example, for the purpose of choosing keys or choosing random nonces. Weak random values may result in an adversary ability to break the system, as was demonstrated by breaking the Netscape implementation of SSL [8], or predicting Java session-ids [11].

Since a physical source of randomness is often too costly, most systems use a pseudo-random number generator. The state of the generator is seeded, and periodically refreshed, by entropy which is gathered from physical sources (such as from timing disk operations, or from a human interface). The state is updated using an algorithm which updates the state and outputs pseudo-random bits.

This paper studies the Linux pseudo-random number generator (which we denote as the LRNG). This is the most popular open source pseudo-random number generator, and it is embedded in all running Linux environments, which include desktops, servers, PDAs, smart phones, media centers, and even routers.

Properties required of pseudo-random number generators. A pseudo-random number generator must be secure against external and internal attacks. The attacker is assumed to know the code of the generator, and might have partial knowledge of the entropy used for refreshing the generator's state. We list here the most basic security requirements, using common terminology (e.g., of [3]). (A more detailed list of potential vulnerabilities appears in [14].)

- *Pseudorandomness.* The generator's output looks random to an outside observer.
- *Forward security.* An adversary which learns the internal state of the generator at a specific time cannot learn anything about previous outputs of the generator.
- *Break-in recovery / backward security.* An adversary which learns the state of the generator at a specific time does not learn anything about future outputs of the generator, provided that sufficient entropy is used to refresh the generator's state.

Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)
1081-6011/06 $20.00 © 2006 IEEE

The *Pseudorandomness* requirement is sufficient in attack scenarios where the adversary does not have knowledge of the generator's state. In many scenarios, however, an adversary might be able to learn the state of the generator; for example, by bypassing access restrictions of the operating system, or by reading the state from memory or from the hard disk if it is stored there. The *forward security* requirement ensures that an adversary which learns the generator's state at a certain time cannot gain information about the output of the generator at previous times (as we show below, this requirement is not fully satisfied by the LRNG). The *break-in recovery* requirement ensures that learning the state at a particular time does not compromise all future outputs of the generator.

1.1 The Linux Pseudo-Random Number Generator (LRNG)

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, on all types of devices.

The output of the LRNG can be used by internal kernel functionalities which use random bits, and by calls to its application programming interface (API). The Linux kernel uses random data for various purposes, such as generating random identifiers, computing TCP sequence numbers, producing passwords, and generating SSL private keys. Within the kernel, the interface for receiving random values from the LRNG is the function `get_random_bytes(*buf, nbytes)`.

The API to the LRNG is through two device drivers named `/dev/random` and `/dev/urandom`. Both devices let users read pseudo-random bits. The difference between the two is in the stated level of security of the random bits, and the resulting delay. The first device (`/dev/random`) outputs "extremely secure" bits[1] and may block the user until such bits can be generated by the system. The second device (`/dev/urandom`) outputs less secure bits but its output is never blocked. Section 2.4 explains the difference between the two devices.

Why reverse-engineering the LRNG is not easy. The LRNG is part of an open source project and therefore one might assume that its source code is available for public scrutiny and that its security can be easily analyzed (or at least, is not based on "security by obscurity"). However, the LRNG is not well documented and there is no clear description of the implemented algorithm. The LRNG is composed of about 2500 lines of code, and in addition, hundreds of code patches were applied to the code during the last five years (and consequently, the available documentation does not always reflect the current code). One example of the complexity of the LRNG code is the fact that for 17 months the LRNG code included a bug in which entropy addition used a vector of size $4 \times n$ instead of n. We also note that throughout our analysis we were not helped by any of the LRNG authors.

These factors turned the reverse-engineering of the LRNG into a challenging task. We therefore combined static reverse-engineering of the source code of the Linux kernel with dynamic tracing to present a clear algorithmic representation of the LRNG (see Section 3). Dynamic reverse-engineering is not simple in this case, due to two main restrictions. The first is the fact that any kernel change requires a new build and installation. This process takes a couple of hours to finish, and performing it dozens of times makes the process very tedious. The second, and more troubling restriction, is the fact that any change made to the kernel may also result in some influence on the kernel "noise generation" and hence on the LRNG behavior. We therefore implemented a user-mode *simulator* of the LRNG as part of our analysis. It can be downloaded from the authors' web page.

As a final note on the complexity of the implementation, we add that the complexity of the algorithm, the lack of documentation, and the high volume of changes to the LRNG code, resulted in dozens of programming bugs. Many of these bugs resulted in security vulnerabilities during the last five years.

The basic structure of the LRNG. At a high level, the LRNG can be described as three asynchronous components. The first component translates system events into bits which represent the underlying entropy. The second component adds these bits to the generator "pool". When bits are read from the generator, the third component applies three consecutive SHA-1 operations to generate the output of the generator and the feedback which is entered back into the pool.

Each sample of "randomness" originating from system events is collected as two 32-bit words. The first word measures the time of the event and the second word is the event value, which is usually an encoding of a pressed key, a mouse move or a drive access. In order to keep track of the amount of physical randomness which is added to the pool, the LRNG holds a counter for counting an estimate of this value, which is calculated as

[1] This wording is according to the LRNG designer. Essentially, this type of output is only available when enough physical entropy is gathered. Our discussion of the resulting security is in Section 3.4.

a function of the frequencies of the different events. The LRNG denotes this value as *entropy* (see Section 2.4 for the exact definition) although it is different than the classical entropy definition by Shannon [21].

1.2 Our Contributions

This paper describes research conducted on analyzing the LRNG. It provides the following contributions:

- Publication of a description of the LRNG algorithm. As described above, a considerable amount of work was required in order to analyze the LRNG code and provide a high-level description of the underlying algorithm.
- An attack which breaks the forward-security of the LRNG. Namely, we show how, given a state of the generator, it is possible to reconstruct previous states. The time complexity of this attack is 2^{64} or 2^{96}, depending on the attack variant. The memory overhead of the attack is $O(1)$.
- An analysis of the amount of entropy which is added to the generator in one typical implementation.
- An analysis of the security of an implementation on a disk-less system (an OpenWRT based router).
- We also identify some vulnerabilities in the current implementation (including an easy deniale of service attack) and provide recommendations for improving future implementations of pseudo-random number generators.

1.3 Related Work

Existing PRNG implementations. In the past, PRNGs were either a separate program or a standard library within a programming language. The evolution of software engineering and operating systems introduced PRNGs which are part of the operating system. From a cryptographic point of view, this architecture has three main advantages: (1) the ability to introduce more complex algorithms which are implemented using unique kernel optimization, (2) the ability to use kernel based entropy events as input to the generator, and (3) the fact that in a multi-user, multi-threaded environment, many consumers can read random bits, and therefore an adversary might be prevented from reading a long stream of consecutive bits of the PRNG output.

A published overview of the use of cryptography in OpenBSD [5] includes a very informative introduction to this operating system's usage of PRNGs, and of cryptography's general role in this operating system. The PRNG of the FreeBSD operating system is described in [18]. It uses time stamps of events as an entropy (i.e., physical randomness) source. These are hashed using AES [19] into two pools, each of 256 bits. When output is extracted, AES encryption is used to repeatedly encrypt a 256-bit counter, using an encryption key that is taken from the entropy pools. FreeBSD implements a single non-blocking device and the authors declare their preference of performance over security.

Castejon-Amenedo et al. [4] propose 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 entropy equally between the two devices, such that there is no entropy that is used in both. A notable advantage of this scheme is the absolute separation between blocking and non-blocking devices, which prevents launching a denial-of-service attack on the blocking device by using the non-blocking device (such an attack is possible in Linux, as we later discuss in Section 3.4).

Analysis of PRNGs. A comprehensive discussion of the system aspects of PRNGs, as well as a guide to designing and implementing a PRNG without the use of special hardware or of access to privileged system services, is given by Gutmann [9]. Issues related to operating system entropy sources were discussed in a recent NIST workshop on random number generation [12, 10].

An extensive discussion of PRNGs, which includes an analysis of several possible attacks and their relevance to real-world PRNGs, is given by Kelsey et al. in [14]. Additional discussion of PRNGs, as well as new PRNG designs appear in [13, 7].

The recent work of Barak and Halevi [3] presents a rigorous definition and an analysis of the security of PRNGs, as well as a simple PRNG construction. This work suggests separating the entropy extraction process, which is information-theoretic in nature, from the output generation process. Their construction is based on a cryptographic pseudo-random generator G, which can be implemented, for example, using AES in counter mode, and which does not use any input except for its seed. The state of the PRNG is the seed of G. Periodically, an entropy extractor uses system events as an input from which it extracts random bits. The output of the extractor is xored into the current state of G. This construction is much simpler than most existing PRNG constructions, yet its security was proved in [3] assuming that the underlying building blocks are secure. We note that our analysis shows that the Linux PRNG construction, which is much more complex than that of [3], suffers from weaknesses which could have been avoided by using the latter construction.

Analysis of open-source security packages. It is hard to examine the security of implementations of security packages, and, in the case of proprietary software, one usually has to trust the software authors. However, even in the case of open source security packages it is hard to trust security, since the fact that source code can be read does not imply that it is actually examined by security experts. For example, the work of Nguyen [20] examined the (open) source code of the GNU Privacy Guard secure email software (GnuPG or GPG), and identified several cryptographic flaws. The most serious of these flaws has been present in GPG for almost four years.

2 The Structure of the Linux Random Number Generator

Our study is based on version 2.6.10 of the Linux kernel, which was released on December 24, 2004.

2.1 General Structure

The generation of random numbers in Linux is composed of three asynchronous procedures. In the first procedure, operating system entropy is collected from various events inside the kernel. In the second procedure, entropy is fed into an LFSR-like pool, using a mixing function. When random bits are requested, the third procedure occurs, output is generated and the pool is updated. The only non-linear cryptographic operation used by these procedures is the SHA-1 hash function. (We note that the recent attacks on the collision resistance of SHA-1 seem irrelevant for the purpose of attacking the LRNG.)

Pools and counters. Figure 2.1 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[2] ; output from the primary pool is extracted and fed to the secondary and urandom pools, while the LRNG output is extracted from the secondary pool or from the urandom pool. During the extraction operation, the inner state of a pool is modified in a feedback manner.

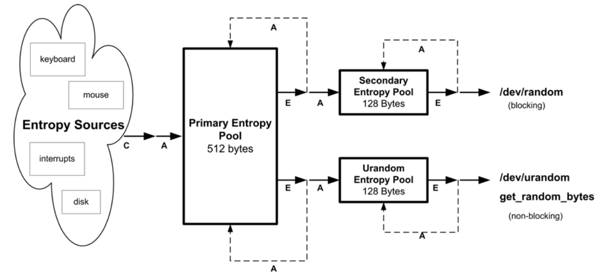

Figure 2.1: The general structure of the LRNG: Entropy is collected (C) from four sources and is added (A) to the primary pool. Entropy is extracted (E) from the secondary pool or from the urandom pool. Whenever entropy is extracted from a pool, some of it is also fed back into this pool (broken line). The secondary pool and the urandom pool draw in entropy from the primary pool.

Each pool has its own entropy estimation counter. This is an integer value between zero and the pool size in bits, which indicates the current estimated entropy of the pool. When output is extracted from the pool this counter is decremented, and when entropy is added the counter is incremented. An entropy counter is always decremented by the number of extracted bits. Incrementing the counter is more complex. If the added bits originate from one of the entropy sources, then their entropy is estimated, and the counter is incremented accordingly. The entropy estimation uses the timing of the last few events of the same entropy source (see discussion below). If the entropy bits are transferred from the primary pool, the entropy counter of the receiving pool is incremented by the number of transferred bits.

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

Adding physical entropy. Entropy bits are added to the primary pool from external sources. Desktop and server PCs can use four different sources: mouse and keyboard activity, disk I/O operations, and specific interrupts. When such an event occurs, it produces a 32-bit word representing its timing and a 32-bit word encoding its attributes (e.g., which key was pressed). In addition,

[2] As is described below, system entropy might also be added to the secondary pool, if the LRNG estimates that the primary pool has full entropy.

the differences between the timings of successive events of the same type are used to estimate the entropy provided by this event. In Section 2.4 we define this procedure in detail.

Due to the asynchronous nature of the system, collected entropy cannot be simply added to the pools but is rather collected and batched. A few times a minute the batched data is added to the pools (in a process described in Section 2.5). The default operation is to add entropy to the primary pool. If this pool is full (its entropy count equals 4096) entropy is added to the secondary pool. When the secondary pool is full the process returns to the primary pool, and so on. Entropy is never added to the urandom pool. This process increments the entropy counter of the respective pool by the estimated entropy amount.

Generating output. Random bits are extracted from one of the three pools: they are extracted 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 other pools does not have enough entropy and needs re-filling. The process of entropy extraction includes three steps: updating the pool's contents, extracting random bits to be output, and decrementing the entropy counter of the pool. This process involves hashing the pool contents using SHA-1, and adding the results to the pool. We study each of the LRNG steps in the following sections.

2.2 Initialization

Operating system startup includes a sequence of routine actions. This sequence includes the initialization of the LRNG with constant operating system parameters and with the time-of-day, and additional disk operations and system events which affect the LRNG using the interface for adding external entropy (discussed in Section 2.5). This sequence of operations might be easily predicted by an adversary, especially in systems which do not have a hard drive. If no special actions are taken, the LRNG state might include very limited entropy. (For example, the time of day is given as a count of seconds and of micro-seconds, each represented as a 32-bit value. In reality these values have very limited entropy as one can find computer uptime within an accuracy of a minute, which leads to a brute-force search of only $60 \times 10^6 < 2^{26}$ different options.)

To solve this problem, the LRNG simulates continuity along shutdowns and startups. This is done by saving a random-seed at shutdown and writing it back to the pools at startup. A script that is activated during system startups and shutdowns uses the read and write capabilities of the `/dev/urandom` interface to perform this operation.

During shutdown the script reads 512 bytes from `/dev/urandom` and writes them to a file, and during startup these bits are written back to the `/dev/urandom` device. This device is defined such that writing to it modifies the *primary* pool and not the urandom pool (as one could expect from its name). The resulting operations applied to the primary pool are pretty much identical to the effect of receiving these 512 bytes as the encoding of system events, and adding them to the primary pool using the usual procedure for adding entropy, which is outlined in Section 2.5. The only difference is that the added bytes do not increment the entropy estimation. The secondary pool and the urandom pool are refreshed by the primary pool, and therefore the script affects all three pools.

It is important to note that this script is part of a Linux distribution package, such as RedHat, and not part of the kernel code itself (this is also the reason that the script must interact with the pools using a device driver rather than reading and writing from/to the pools). The author of the LRNG ([22]) instructs Linux distribution developers to add this script in order to ensure the unpredictability of the LRNG at system startups. This implies that the security of the LRNG is not completely stand-alone, but dependent on an external component, which can be predictable in certain Linux distributions.

Security implications: Some Linux distributions, such as the KNOPPIX distribution which is a bootable PC system on a CD or a DVD [1], or the OpenWRT Linux distribution for routers [2], do not use a script of this type and therefore initialize the LRNG from scratch in each reboot. This might result in an initial LRNG state which is rather predictable. It is also obvious that when the seed is saved in a file on the hard disk after system shutdown, anyone who can physically access the disk can read that file and learn the seed, or alternatively replace the seed with a different value (say, a previous value of the seed for which the adversary already recorded the output of the generator). Access permissions are not too helpful here, since they are related to the operating system and not to the hard disk.

2.3 Collecting Entropy

In a PC environment, the LRNG collects entropy from events originating from the keyboard, mouse, disk and system interrupts. When such an event occurs, two

Keyboard	Mouse	Hard Drive	Interrupts
8	12	3	4

Table 1. The number of unknown bits in operating system events.

32-bit words are used as input to the entropy pools. The first word encodes the timing of the event in *jiffies* (namely, the number of milliseconds from the time the machine was booted) or in cpu-cycles granularity (currently cpu-cycles granularity is only used on SMP). The second word encodes the event type. For example, in case of a keyboard event the word encodes the key that was pressed. Table 1 presents the number of unknown bits per each type of event. Note that the actual entropy of these events is much lower, as most of them are predictable to a large extent. Appendix A describes in detail how each value is calculated.

In other environments, the LRNG gathers entropy from the available resources. For example, an OpenWRT router does not include a hard disk, mouse and keyboard and therefore these cannot be used as an entropy sources. On the other hand, the router collects entropy from network events.

2.4 Estimating the Entropy Amount

One of the fundamental issues in using physical entropy is estimating the amount of entropy which is added to the generator, and more generally, estimating the current amount of entropy in the pools.

As noted above, the difference between `/dev/random` and `/dev/urandom` is that the `/dev/random` interface does not return more bits than its current entropy estimation and thus might block. The `/dev/urandom` interface, and the kernel interface (`get_random_bytes`), return any number of pseudo-random bits, according to the request. This difference implies that entropy estimation is important mainly for the `/dev/random` interface.

The LRNG estimates the amount of entropy of an event as a function of its timing only, and not of the event type. The estimate is done in the following manner:

Definition 2.1 *Let t_n denote the timing of event number n. Define*

$$\delta_n = t_n - t_{n-1}$$
$$\delta_n^2 = \delta_n - \delta_{n-1}$$
$$\delta_n^3 = \delta_n^2 - \delta_{n-1}^2$$

Note that t_n, δ_n, δ_n^2, δ_n^3 are each 32 bits long.

The amount of entropy added by the event is defined to be $\log_2\left(min\left(|\delta_n|,\ |\delta_n^2|,\ |\delta_n^3|\right)_{[19-30]}\right)$ where $S_{[a-b]}$ denotes bits a to b (inclusive) of S (where location 0 is the MSB). If $min\left(|\delta_n|,\ |\delta_n^2|,\ |\delta_n^3|\right)_{[19-30]} = 0$ then the estimate is 0. (Even if the estimate is 0, the event is used to update the state of the LRNG. The entropy count, however, is only updated if the entropy estimate of the event is positive.)

This estimation is relevant only in the case of adding entropy from OS sources to the pools. When a user writes data to one of the device drivers (`/dev/random` or `/dev/urandom`) the entropy counter is not changed. When n bits are extracted from a pool the entropy estimation is decremented by n. When bits are transferred from one pool to another the first is decremented and the second incremented—both by the amount of transferred bits.

2.5 Updating the Pools

The mechanism for updating the pools is based on a TGFSR (Twisted Generalized Feedback Shift Register [15, 16]). The main advantage of the TGFSR is its extended cycle length for any initial seed. The period of a TGFSR with a state of 128 words (on a 32-bit PC) can be $2^{128\times 32} - 1$ steps.

Definition 2.2 (TGFSR) *A series $a_i \in \{0,1\}^w$ and a matrix $A_{w\times w}$ are a TGFSR based on a primitive polynomial $x^p + x^{p-t_1} + \cdots + x^{p-t_m} + 1$ ($1 \le t_1, \ldots, t_m < p$) if and only if*

$$\begin{aligned} a_i = a_{i-p+t_1} \oplus \cdots \oplus \quad & a_{i-p+t_m} \oplus a_{i-p}A \\ & i = p, p+1, \ldots \end{aligned}$$

The only input to the TGFSR is the initial value of the state (which is a $p \times w$ bit seed), and in each iteration the internal state is used to generate the new state.

The shift register used in the LRNG is based on the TGFSR (and its implementation which is described in [15]) but is different from it since it adds entropy in each iteration. We define the pools of the LRNG as arrays of length m words ($m = 32$ or $m = 128$) which are indexed by an index j.

Adding entropy. Entropy is added in each round of state and output computation, as well as when entropy is added from external sources, or from the primary pool to the secondary and urandom pools. Entropy is added by running the algorithm `add(pool,j,g)` and updating the value of the index j, where g is the new entropy word which is added to the pool. Figure 2.2 defines the update operation for a pool of size 32 words.

```
Algorithm add(pool, j, g):
   temp := g
   temp := temp xor pool[j]
   temp := temp xor pool[(j+1) mod 32]
   temp := temp xor pool[(j+7) mod 32]
   temp := temp xor pool[(j+14) mod 32]
   temp := temp xor pool[(j+20) mod 32]
   temp := temp xor pool[(j+26) mod 32]
   temp := (temp >> 3) xor table[temp & 7]
    // the last 3 bits of temp choose a table
    // entry which is xored to (temp >> 3)
   pool[j] := temp

   // table[] is defined as follows
   // table[0] = 0x0
   // table[1] = 0x3b6e20c8
   // table[2] = 0x76dc4190
   // table[3] = 0x4db26158
   // table[4] = 0xedb88320
   // table[5] = 0xd6d6a3e8
   // table[6] = 0x9b64c2b0
   // table[7] = 0xa00ae278
```

Figure 2.2: Pseudo-code of the entropy addition algorithm for the urandom and secondary pools. *pool* is the pool to add entropy to, *g* is the added entropy, *table* is a table with eight words, and *j* is the current position in *pool*.

Each pool is updated based on a primitive polynomial. The polynomial is chosen according to the size of the pool, and therefore the secondary and the urandom pools share the same polynomial, which is $x^{32} + x^{26} + x^{20} + x^{14} + x^7 + x + 1$. The primary pool polynomial is $x^{128} + x^{103} + x^{76} + x^{51} + x^{25} + x + 1$, and the entropy addition for that pool is identical to that of the smaller pools, except for using this polynomial for updating the pool (namely, xoring g with entries $j, j+1, j+25, j+51, j+76$ and $j+103$ modulo 128).

Entropy addition can be analyzed by assuming that the generator is reseeded in each iteration. Alternatively, one might analyze it by assuming that the TGFSR is used to encrypt the entropy input. The LRNG reseeding process changes the elementary properties of the TGFSR. The long period can no longer be guaranteed, and the process is no longer a linear function of the initial state.

2.6 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`. It is also extracted from the primary pool for the purpose of refreshing the other pools.

Extracting entropy from a pool is not a simple operation. It involves hashing the extracted bits, modifying the pool's state and decrementing the entropy estimate by the number of extracted bits.

Figures 2.3 and 2.4 present a pseudo-code and a diagram of the extraction algorithm. Entropy extraction is done in blocks of 10 bytes. The process is described for the case of the urandom or secondary pools, which are 32 words long. For simplicity the description does not include the steps of decrementing the entropy estimation, and the entropy refilling process. The algorithm applies the SHA-1 function to the first 16 words, and adds part of the result to location j. It then applies a variant of SHA-1, which we denote as SHA-1' (see below), to the right half of the pool, and adds parts of the result to locations $j-1$ and $j-2$. Finally, it applies SHA-1' to the 16 words ending at location $j-2$, and uses the result to compute the output in the following way: The output of SHA-1' is 5 words (20 bytes) long. These words are folded as described in Figure 2.5, and the resulting 10 bytes are the output of the iteration. This output is 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.

```
Algorithm Extract(pool, nbytes, j):
   while nbytes > 0
         tmp := SHA−1(pool[0..15])
                // the result is 5 words long
         add(pool, j, tmp[0])
         tmp := SHA−1'(pool[16..31])
         add(pool, j−1 mod 32, tmp[2])
         add(pool, j−2 mod 32, tmp[4])
         tmp := SHA−1'(pool[(j−2−15) mod 32
                    ... (j−2) mod 32])
         tmp := folding(tmp[0..4])
                // the result is 2.5 words long
         output(tmp, min(nbytes, 10))
         nbytes:=nbytes−10
         j := j−3 mod 32
   end while
```

Figure 2.3: Pseudo-code of the extraction algorithm. *pool* is a 32 work pool from which entropy is extracted, *nbytes* is the number of requested bytes, and *j* is the current position in *pool*.

The SHA-1 variant being used. The first hash function used in the procedure is the original SHA-1 function (see [6] for the exact definition of SHA-1). Each of the following hash operations in this iteration, which we denote as SHA-1', use for their five initial constant values (IV) the five output words of the previous hash result. (We note that if the LRNG had used the original SHA-

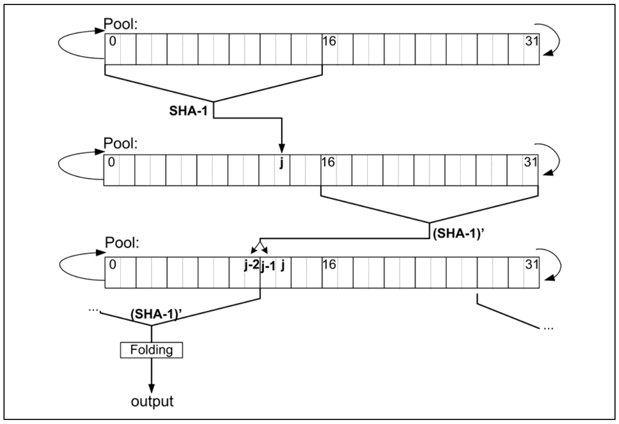

Figure 2.4: Extraction algorithm

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

Figure 2.5: Folding operation. Folding 5 words (160 bits) to 2.5 words (80 bits). W_i denotes the i^{th} word, $W_{i_{[l-m]}}$ denotes bits $l - m$ (inclusive) of the i^{th} word..

1 function, the attacks in Section 3.1 would have been considerably more efficient, as is described there.)

Extracting randomness from the primary pool. Bits are extracted from the primary pool when it is required to refresh one of the other pools. In this case, the algorithm is slightly different since the primary pool is longer. The SHA-1 (or SHA-1') operation is applied to each of the eight 16 word chunks of the primary pool, and once more to the 16 words ending at location $j - 8$. The first eight SHA-1 operations update eight words in the pool, and the last operation generates the output. The algorithm is described in Appendix B.

3 Analysis

Our analysis of the security of the LRNG is composed of four parts: (1) a cryptanalytic attack on the forward security of the LRNG, (2) an analysis of the entropy added by system events, (3) observations on the insecurity of the LRNG in the OpenWRT Linux distribution for routers, and (4) observations on security engineering aspects of the LRNG, including a denial-of-service attack.

3.1 Forward Security

We first observe that the output which is extracted from a pool is calculated as the last state in the `Extract` algorithm. Namely, it is computed *after* the state of the pool is updated. This means that if the state of the pool at time t is known then it is easy to compute the output which was extracted from the pool during its last state transition (i.e., the output which was computed in the transition from time $t - 1$ to time t). This is a flaw in the forward security of the LRNG, since it enables anyone who observes the state of the LRNG at a certain time to compute the last output of the LRNG. We show below how to mount a stronger attack on the forward security of the LRNG and compute, given the state at time t, previous states, which consequently enable to compute previous outputs of the LRNG.

We describe below how to reverse the state of a single pool, assuming that in its last update it was not refreshed with new entropy. Let us recall that the states of the urandom and secondary pools are updated when random bits are extracted from the LRNG, and that if the entropy estimates of these pools are low these pools attempt to be refreshed with output from the primary pool. Only the secondary pool, however, is blocked if no such refresh is available. The state of the primary pool is updated with randomness from system events whenever it is updated. The attack is therefore mostly relevant to the urandom pool which is often used for extracting many bits while not receiving any entropy updates. The attack is also relevant to the secondary pool, if the attack starts from a time in which the value of the entropy counter is high, and to the primary pool, if the entropy which is added to the pool is mostly predictable.

The input of the attack is the state of a pool at time t (denoted as $\mathrm{pool_t}$). It computes the state of the pool at time $t - 1$ (denoted as $\mathrm{pool_{t-1}}$). Now, given $\mathrm{pool_{t-1}}$ it is easy to compute the random value which was output in the extract operation that transitioned the pool state from $\mathrm{pool_{t-2}}$ to $\mathrm{pool_{t-1}}$. In other words, the *forward security* requirement is not satisfied since it is possible to compute a previous output of the LRNG. The same analysis can be continued and compute the state and the corresponding outputs at times $t-2, t-3$ and so on, until the last time that the pool received an entropy update.

Below we describe two methods for reversing the state of the LRNG. The first method is a generic attack which has an overhead of 2^{96}, which is much better than an exhaustive search (whose overhead is 2^{1024} for the case of a 32 word pool), but is rather impractical. The second attack is almost practical, with an overhead of 2^{64}, but it is only applicable when the index j is in a specific range (covering 18 of the 32 possible values of j in a 32 word pool). This means that a lucky attacker, which starts its attack when the value of j is at the end of this range, can compute five previous states of the pool (and the corresponding outputs) with an overhead of about 2^{64}. Both attacks use $O(1)$ memory.

The `Extract` algorithm is described in Figure 2.3 and is used for advancing the pool and extracting an output from it. To simplify the analysis, let us first assume that the pool is either the secondary or the urandom pool, and is therefore 32 words long, and that the `add` operation is a simple addition modulo $2^{32} - 1$, instead of the TGFSR operation detailed in Section 2.5.

The analysis starts with knowledge of the state of the pool at time t (namely, $\mathrm{pool_t}$), and of the value of the index j. (To simplify the notation, we denote by j_t, or simply by j, the value of the index j at the *beginning* of the computation of the `Extract` operation that transitions the pool from time $t-1$ to time t.)

A generic attack: As a warmup we describe a generic attack which is based on a simple observation: all but three words of the pool (those indexed by $j, j-1$ and $j-2$) are identical in both $\mathrm{pool_t}$ and $\mathrm{pool_{t-1}}$. Given $\mathrm{pool_t}$ there are therefore only 2^{96} possible candidate values, or "guesses", for $\mathrm{pool_{t-1}}$ — those obtained by copying the values of the words in locations different from $[j-2, j]$ and going over all possible values for the 96 bits in words $j, j-1$ and $j-2$. The transition from $\mathrm{pool_{t-1}}$ to $\mathrm{pool_t}$ is deterministic and defined by the `Extract` algorithm in Figure 2.3. The attack therefore applies this algorithm to each candidate value of $\mathrm{pool_{t-1}}$, and checks if the result is equal to $\mathrm{pool_t}$. If the two values are not equal then the candidate value is dismissed. Otherwise, the candidate value is put in a short list of possible values for $\mathrm{pool_{t-1}}$ (we show immediately that this list is indeed short).

Note that although the `Extract` procedure uses three invocations of SHA-1, which cause the bulk of the overhead, all but a fraction of 2^{-32} of the candidates for $\mathrm{pool_{t-1}}$ can be dismissed after a *single* application of SHA-1. The overhead of the search is therefore about 2^{96} applications of SHA-1.

Estimating the accuracy of the attack. We know that the true value of $\mathrm{pool_{t-1}}$ is in the computed short list, but that there might be some additional "false positives". I.e., values for locations $[j-2, j]$ in time $t-1$ which are different from the true value, but for which applying the `Extract` algorithm results in the right value for $\mathrm{pool_t}$. Fortunately, we do not expect many false positives. There are $2^{96} - 1$ false candidates for $\mathrm{pool_{t-1}}[j-2, j]$, and each of them has a probability of 2^{-96} to become a false positive (assuming that we model SHA-1 as a random function and therefore the probability of computing the right value of $\mathrm{pool_t}[j-2, j]$ is 2^{-96}). The number of false positives is therefore k with probability of about $\binom{n}{k} n^{-k} (1 - 1/n)^{n-k}$, where $n = 2^{96} - 1$. Namely, with probability e^{-1} there are no false positives ($k = 0$), with probability e^{-1} there is a single false positive, with probability of about $0.5e^{-1}$ there are two false positives, and so on.

Given the short list of possible values of $\mathrm{pool_{t-1}}$, the previous procedure can be applied for each value in the list to compute all possible values of $\mathrm{pool_{t-2}}$. Applying this procedure to the correct value of $\mathrm{pool_{t-1}}$ always results in one or more candidates for $\mathrm{pool_{t-2}}$, which include the correct value of $\mathrm{pool_{t-2}}$ and possibly additional false positives (according to the distribution that was detailed above). However, applying the procedure to a value which is a false positive for $\mathrm{pool_{t-1}}$, results, with probability e^{-1}, with no candidate for $\mathrm{pool_{t-2}}$. If this event happens then it can be concluded that the tested value for $\mathrm{pool_{t-1}}$ is a false positive, and this value can be removed from the list. (If this is not the case then with probability e^{-1} the tested value results in a single candidate for $\mathrm{pool_{t-2}}$, with probability $0.5e^{-1}$ it results in two candidates for $\mathrm{pool_{t-2}}$, etc.) The number of possible values for $\mathrm{pool_{t-k}}$ is therefore a random variable. In Appendix C we show that the sequence $\{|\mathrm{pool_{t-k}}| - k\}_{k=1,2,\ldots}$ is a martingale, that $E(|\mathrm{pool_{t-k}}|) = k$, and that the probability that $\mathrm{pool_{t-k}} > k + b$ is at most $1/b$. We can therefore conclude that for all practical purposes the procedure outputs a list of reasonable size of the possible values of the state at time $t-k$.

A more efficient attack. We now show that for 18 of the possible 32 values of the index j, it is possible to reverse the pool by a procedure with an overhead of 2^{64}. This procedure is applicable if the value of j is in the range $[16, 31]$, and for $j = 1, 2$. We detail the procedure for the case of $j \in [18, 31]$. In this case the words which are affected by the state transition are located in the upper half of the pool, while the first half of the pool does not change from time $t-1$ to time t (namely, $\mathrm{pool_t}[0, 15] = \mathrm{pool_{t-1}}[0, 15]$). It is therefore possible, given $\mathrm{pool_t}$, to apply SHA-1 to words $[0, 15]$ and compute the value that was added to location j. Given this value it is possible to compute $\mathrm{pool_{t-1}}[j]$ from $\mathrm{pool_t}[j]$. In addition, the initialization vector for the second application of SHA-1 is computable from $\mathrm{pool_{t-1}}[0, 15]$. It is therefore possible to go over all 2^{64} potential values of $\mathrm{pool_{t-1}}[j-2, j-1]$, apply SHA-1 to $\mathrm{pool_{t-1}}[16, 32]$ and compute the resulting values that were added to locations $j-2$ and $j-1$. If these values are not equal to the difference between the values of these locations in time t and in time $t-1$, it is safe to dismiss the "guess" of $\mathrm{pool_{t-1}}[j-2, j-1]$. The true value of $\mathrm{pool_{t-1}}$ is never dismissed, while false positives appear with the same probability distribution as in the previous analysis (the only difference is the value of n, which is 2^{64} instead of 2^{96}). The number of possible candidates for $\mathrm{pool_{t-k}}$ behaves according to the same distribution as in the previous procedure, as is analyzed in Appendix C. The expected number of candidates for $\mathrm{pool_{t-k}}$ is therefore only k. The overhead of the attack is $O(2^{64})$ operations and $O(1)$ memory.

We note here that the pool could have been reversed with an overhead of 2^{64} or less operations for *any* value of j, if the `Extract` procedure had used the original SHA-1 function rather than the version which changes the initialization vector after the first invocation of the function. The effect of the SHA-1 variant which is used in the LRNG is that one cannot compute the values added to locations $j-1$ and $j-2$ before computing $\mathrm{pool_{t-1}}[0, 15]$. For $j \in [3, 15]$ this means that we must check all options for $\mathrm{pool_{t-1}}[j-2, j]$.

Reversing the primary pool. Assume that we are given the state of the primary pool, and that we know which entropy value was added to the pool when it was advanced to this state. The only difference of this case from the case of the shorter pools is that the size of the pool is 128 words. The generic attack can still be applied with a 2^{96} overhead. The more efficient attack can be applied if j is in the range $[120, 127]$.

The `add` operation. The previous analysis assumed that the `add` operation in the `Extract` algorithm is an addition modulo $2^{32}-1$. However, this operation is implemented as is described in Section 2.5 and therefore the value added to location j is a function of the current values of several locations in the pool, as is depicted in Figure 2.2. This change obviously does not affect the generic attack, since $\mathrm{pool_t}$ is still a deterministic function of $\mathrm{pool_{t-1}}$. The second, more efficient, attack is also not affected: the value added to location j is a function of the result of applying SHA-1 to $\mathrm{pool_{t-1}}[0, 15] = \mathrm{pool_t}[0, 15]$ and computing a function of the SHA-1 result, of $\mathrm{pool_{t-1}}[j]$, and of five other locations in $\mathrm{pool_{t-1}}$ which are not changed in the transition to $\mathrm{pool_t}$. Given $\mathrm{pool_t}[j]$, we can examine its three most significant bits and identify the entry of `table` which was xored to `temp` in the transition (this is easy since each of the eight entries in `table` has a different value to its three most significant bits). The index of the entry identifies the three least significant bits of `temp` (before it was shifted to the right). It is now possible to reverse the operation that was performed in the second to last line of `add`, and xor the result with `g` and with the five pool entries which where used to generate it. The result is $\mathrm{pool_{t-1}}[j]$.

3.2 Entropy Measurements

When entropy is added to the LRNG pool, each event adds two 32 bit words. In Section 2.3 we described the actual active range of bits within the **type-value** field and Appendix A provides the details for each case.

We now turn to the entropy of the timing of the encoded events. As mouse and keyboard events are usage dependent and network interrupts are not commonly used as an entropy resource for the LRNG, we ran a trial which measured the entropy which is added to the LRNG by hard disk events. Our trial included over ten days of measurements on a single Linux machine, with a total of over $140,000$ entropy addition events. The system was mostly idle, and the measurements were in a setting that recorded their results on a different computer, without affecting the disk of the examined machine.

We mark by t_n the time of event n, and the difference between two consecutive events is defined as $\delta_n := t_n - t_{n-1}$. Table 2 presents the frequency of different δ_n values. Note that all but 0.8% of the values are in the range $[77, 82]$. All other values are larger by about four orders of magnitude. The resulting entropy is only $H := 1.03$ bits per event, which is to say that event timing, which is encoded as a 32 bit field, had in this setting an entropy of at most a single bit. The recordings furthermore show that there is a correlation between consecutive δ_n values (and therefore the entropy is even lower). The entropy of pairs of events is 1.53 bits per pair, and consequently, the conditional entropy of δ_n given δ_{n-1} is only 0.5 bits.[3]

We also checked the entropy estimate that would have been calculated by the LRNG for these measurements, using the procedure outlined in Section 2.4. This estimate turns out to be very conservative. Only 2224 of the 140,000 measurements resulted in a positive addition to the entropy estimate. These events occurred, more or less, only when δ_n had a very large value, and consequently the n and $n+1$ measurements had a large entropy estimate (namely, the 1100 events of Table 2 for

[3] Closer examination reveals patterns such as the following one: For about 84% of the measurements it holds that $\delta_{i+1} = \delta_i$. Given this event, the conditional probability that $\delta_{i+2} = \delta_i$ is about 90%, and the conditional probability that $\delta_{i+2} \neq \delta_i$ but $\delta_{i+3} = \delta_i$ is 9.9%. Consequently, for only 0.1% of the cases in which $\delta_{i+1} = \delta_i$ we get that both δ_{i+2} and δ_{i+3} are different from δ_i.

δ_n	Frequency
77	243
78	1730
79	11468
80	113402
81	11786
82	625
83	0
84	0
> 84	1112

Table 2. Frequency of time differences between two consecutive HD entropy addition

which $\delta_n \gg 84$, each resulted in a pair of consecutive measurements which contributed to the entropy count). The total estimate of the added entropy was 17500 bits. Given that the measurements were taken over a period of ten days, there was an average delay of about 15 minutes between pairs of events which had a positive contribution to the entropy count. The average contribution of these pairs was about 16 entropy bits. This is quite a severe bottleneck for the blocking interface to the LRNG.

3.3 Analysis of the OpenWRT Linux Distribution

OpenWRT [2] is a Linux distribution for wireless routers. It provides many cryptographic services such as SSL termination, a SSH server, and a wireless encryption. The security of all these services is dependent upon the output of the LRNG.

An OpenWRT router has very limited entropy sources. There are no keyboard, mouse or hard drive attached to the router. The WRT uses its flash memory (of size $4 - 16$ MBytes) as a special file system, but this file system does not provide entropy to the LRNG.

In addition, the WRT implementation does not save any LRNG state between reboots. Hence, the only entropy source for the LRNG is network interrupts, which might be observable by an adversary (!). This is true in particular for wireless routers where most network interrupts are caused by wireless activity, which is easily visible by an external adversary.

Given this data we can conclude that the WRT implementation of the LRNG is weak. The state of the LRNG is reset in every reboot to a predictable value (composed of the time of day and a constant string), and the only source of entropy is, to a large extent, observable by external adversaries. The result is that an adversary can simulate the state and the output of the LRNG. We note that security can be improved by saving LRNG output at shutdown and loading it into the state at reboot. This would require potential attackers to either eavesdrop to all network traffic from the time the router was initialized, or obtain access to the LRNG state.

3.4 Security Engineering

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 its output when the entropy estimate is low, until additional "noise" is added to the pools. These facts together suggest two denial of service attacks which 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 results in blocking other users, and might delay them for a long period of time. We tested this attack using simple `dd if=/dev/random` and were able to block other readers of `/dev/random`.

Furthermore, an attack can even be mounted remotely, by triggering system requests for random bytes (`get_random_bytes`) in a significantly higher rate than that of the entropy input events. Since the urandom non-blocking pool (from which these bits are taken) is refilled from the (blocking) primary pool, this attack will result in denial-of-service for the primary and secondary pools. 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 valuable resource, its consumption must be controlled. The common solution in operating systems for such a resource is through the definition of a new quota per user or group for the consumption of random bits.

Guessable passwords. Usually, the first user-operation in a computer system is user login, and the first input entered by the user is the password, or a user-name and password pair.

In a scenario of a disk-less system, without a random seed that is saved between startups, we can imagine a situation where an attacker knows the initial state of the LRNG, and where the sequence of updates of the LRNG during system reboot is quite predictable. The state of the LRNG might therefore be, to a large extent, a deterministic function of the initial password entered by the user. If the attacker is able to read random bits fast enough, it might be able to identify the password by going over all possible password values, and checking which one results in the LRNG output which was observed.

This attack, which was noted by Kelsey et al. [14], is particularly relevant if the LRNG does not obtain input from a hard disk or from interrupts resulting from network events (as is the case for example with one of the most popular network cards, the 3Com PCI 3c905B Cyclone 100baseTx). In this case the input to the LRNG comes mostly from the user's input. We attempted to use this observation to extract the user password from the output of the LRNG of the KNOPPIX Linux distribution [1], which is bootable from a CD and therefore does not save the LRNG state. We were unsuccessful in this attack, largely because the examined system used a hard disk which provided considerable amount of entropy during the boot process.

Solution. It is better to remove the influence of the values of keyboard events on the LRNG. Keyboard entropy should be based on the timing of its events, and not on the type-values. (The timing of keyboard events might also reveal information about data entered by the user, but this seems like a second-order risk compared to the risk from the values of keyboard events.)

An adversary can create noise that directly affects the LRNG output. Normally, there is a separation between the input and output of the LRNG, but when the primary pool is full, the batched entropy is added directly to the secondary pool, from which it is output when `/dev/random` is used. This direct flow between the input and the output of the LRNG might provide an adversary with the ability to create noise that directly affects the generator's output.

Solution. It is best to 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.

The LRNG state reveals the previous LRNG output. The `Extract` procedure (Fig. 2.3) first updates the pool and then computes its output. The result of this design decision is that an adversary which learns the internal state of the LRNG learns the state of the pool which was used to compute the last LRNG output, and can easily compute this output (and hence break the forward-security of the LRNG).

Solution. It is best to switch the order of operations. The state update will than take place after LRNG output.

4 Conclusions and Recommendations

This paper analyzes the Linux random number generator. The LRNG algorithm is complex and includes a large state made of three different storage pools, a complex mechanism for adding entropy from system events, and an extraction algorithm based on a shift register and several SHA-1 operations.

We showed that these layers add complexity to the implementation but do not prevent attacks on the forward security of the LRNG. In addition we described weaknesses in the OpenWRT Linux distribution.

Our study was conducted on the latest (at the time) Linux kernel, labeled version 2.6.10, which was released on December 24, 2004. Since then the kernel kept developing. Lately, version 2.6.15 was released in January 2006, and patches are being published since then[4].

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

Our study does not cover all Linux kernel options. For example, we did not take into account multi-cpu hardware configurations, or unique hardware configurations such as the Qtronix keyboard and mouse device[5] whose entropy collection method is different than the one described here.

Open source security. The LRNG is an open source project which enables an adversary to read the entire source code and even trace changes inside the source configuration management system. This feature gives powerful tools to the adversary. On the other hand, open source benefits security by enabling security audits, and enabling easy changes to the code. It is rather easy to add patches to the current LRNG code in order to prevent the attacks we described in this paper (this would have been much harder, if at all possible, for closed source PRNGs).

[4] See `http://www.linuxhq.com/kernel/file/drivers/char/random.c` for the incremental changes in `random.c`.

[5] `http://lxr.linux.no/source/drivers/char/qtronix.c?v=2.6.10`

"Open" is not a synonym for "secure". We feel that the open source community should have a better policy for security sensitive software components. These components should not be treated as other source elements. We suggest to add a better quality assurance procedure for the cryptographic elements of the kernel. For example, the PRNG must pass statistical tests which can be put into the kernel build process. Open source must also have, in our opinion, a clear and updated documentation of the algorithms used in the code. Such documentation could have saved us from the trouble of reverse engineering the code, and would have provided better access for other researchers to review the security of the LRNG.

4.1 Recommendations

Following our analysis of the LRNG, we suggest the following recommendations for the design of pseudo-random number generators.

- *Fixing the LRNG.* The issues which were reported in this paper should be fixed. In particular, the LRNG code should be changed to prevent attacks on its forward security. The OpenWRT implementation should be changed to provide more entropy to the LRNG, or at least save its state during shutdown.

- *Implementing a quota for the consumption of random bits.* Random bits are a limited resource, and attackers can easily mount a denial-of-service attack (even remotely) by consuming random bits at a high rate. The common solution for this type of problem is to implement a quota system which limits the effect of each user, or each process, on the operation of other users of the same system. Such a quota system should be added to the Linux kernel.

- *Adopting the Barak-Halevi construction.* The Barak-Halevi (BH) construction and its analysis [3] are attractive in their simplicity, which clearly identifies the role of every component of the system, and enables a simple implementation. In comparison, the current LRNG construction is an overkill in some aspects (like the size of the pools or the number of SHA-1 invocations), but its complexity does not improve its security but rather hides its weaknesses. We suggest that future constructions of pseudo-random number generators follow the BH construction (and in general, try to "keep it simple").

- Since randomness is often consumed in a multi-user environment, it makes sense to generalize the BH model to such environments. Ideally, each user should have its own random-number generator, and these generators should be refreshed with different data which is all derived from the entropy sources available to the system (perhaps after going through an additional PRNG). This architecture should prevent denial-of-service attacks, and prevent one user from learning about the randomness used by other users.[6]

References

[1] http://www.knoppix.org/.

[2] `http://openwrt.org`.

[3] B. Barak and S. Halevi. An architecture for robust pseudo-random generation and applications to /dev/random. In *Proc. ACM Conf. on Computing and Communication Security (ACM CCS)*, 2005.

[4] J. Castejon-Amenedo, R. McCue, and B. Simov. Extracting randomness from external interrupts. In *The IASTED International Conference on Communication, Network, and Information Security*, pages 141–146, USA, December 2003. IASTED.

[5] T. de Raadt, N. Hallqvist, A. Grabowski, A. D. Keromytis, and N. Provos. Cryptography in openBSD: An overview. In *USENIX Annual Technical Conference, FREENIX Track*, pages 93–101. USENIX, 1999.

[6] D. Eastlake and P. Jones. US secure hash algorithm 1 (SHA1). Request for Comments 3174, Internet Engineering Task Force, Sept. 2001.

[7] N. Ferguson and B. Schneier. *Practical Cryptography*. John Wiley & Sons, 2003.

[8] I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr Dobb's*, pages 66–70, January 1996.

[9] P. Gutmann. Software generation of practically strong random numbers. In *Proc. of 7th USENIX Security Symposium*, 1998. An updated version appears in `http://www.cypherpunks.to/~peter/06_random.pdf`.

[6] In the current centralized system this property is not guaranteed. A user which learns the state of the generator can simulate future outputs of the generator (assuming that no external entropy is added). If the user later reads random bits it can identify their location in the simulated generator output and learn the values used by other users.

[10] P. Gutmann. Testing issues with os-based entropy sources. `http://www.cs.auckland.ac.nz/~pgut001/pubs/nist_rng.pdf`, July 2004.

[11] Z. Gutterman and D. Malkhi. Hold your sessions: An attack on Java session-id generation. In A. J. Menezes, editor, *CT-RSA*, pages 44–57. RSA, February 2005.

[12] J. Kelsey. Entropy and entropy sources in x9.82. `http://csrc.nist.gov/CryptoToolkit/RNG/Workshop/EntropySources.pdf`, July 2004.

[13] J. Kelsey, B. Schneier, and N. Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Selected Areas in Cryptography*, LNCS vol. 1758, pages 13–33. Springer, 1999.

[14] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption*, LNCS vol. 1372, pages 168–188. Springer, 1998.

[15] M. Matsumoto and Y. Kurita. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation*, 2(3):179–194, 1992.

[16] M. Matsumoto and Y. Kurita. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation*, 4(3):254–266, 1994.

[17] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[18] M. R. V. Murray. An implementation of the Yarrow PRNG for FreeBSD. In S. J. Leffler, editor, *BSDCon*, pages 47–53. USENIX, 2002.

[19] National Institute of Standards and Technology (NIST). Advanced Encryption Standard. Available on: `http://csrc.nist.gov/CryptoToolkit/aes/`.

[20] P. Q. Nguyen. Can we trust cryptographic software? cryptographic flaws in gnu privacy guard v1.2.3. In *EUROCRYPT '2004*, LNCS vol. 3027, pages 555–570. Springer, 2004.

[21] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4), Oct. 1949.

[22] T. Ts'o. `random.c`—Linux kernel random number generator. `http://www.kernel.org`.

A Entropy Collection

We explain each "noise" source and the different valid values for the 32 bits of the `type-value` during the entropy addition procedure:

keyboard event. The type-value contains the keyboard press and release codes, valid range between $[0, 255]$.

mouse event. type-value $:= (\mathit{type} \ll 4) \oplus \mathit{code} \oplus (\mathit{code} \gg 4) \oplus \mathit{value}$. 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 the axis of the movement (horizontal or vertical) in case of mouse movement; *value* is true/false in case of mouse buttons press or release[7], 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) I/O operation. Its type-value is composed of *major* and *minor* numbers (*major* and *minor* numbers are operating system symbols that together uniquely define a device):

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

If there is only one IDE disk, the type-value is fixed, since the major and minor numbers are constants. (In most cases the major is 3 (first IDE disk) and the minor is 0 (master), and their combined type-value yields $0x300100$.) 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 values to the LRNG. In many setups interrupts will not add any entropy events.

[7] 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 $value = 0$.

B Extracting Randomness from the Primary Pool

The primary pool is updated using a procedure which is described in the following pseudo-code:

```
Algorithm Extract(pool, nbytes, j):
    while nbytes > 0
            tmp := SHA−1(pool[0..15])
// the result is 5 words long
            add(pool, j, tmp[0])
            tmp := SHA−1'(pool[16..31])
            add(pool, (j−1) mod 128, tmp[2])
            tmp := SHA−1'(pool[32..47])
            add(pool, (j−2) mod 128, tmp[4])
            tmp := SHA−1'(pool[48..63])
            add(pool, (j−3) mod 128, tmp[1])
            tmp := SHA−1'(pool[64..79])
            add(pool, (j−4) mod 128, tmp[3])
            tmp := SHA−1'(pool[80..95])
            add(pool, (j−5) mod 128, tmp[0])
            tmp := SHA−1'(pool[96..111])
            add(pool, (j−6) mod 128, tmp[2])
            tmp := SHA−1'(pool[112..127])
            add(pool, (j−7) mod 128, tmp[4])
            add(pool, (j−8) mod 128, tmp[1])
            tmp := SHA−1'(pool[(j−8) mod 128
                          ... (j−8−15) mod 128])
            tmp := folding(tmp[0..4])
// the result is 2.5 words long
            output(tmp, min(nbytes, 10))
            nbytes:=nbytes−min(nbytes, 10)
            j := (j−9) mod 128
    end while
```

C Probability Calculations

The number of false positives generated by the procedure for reversing the pool is a random variable with the following distribution: The probability of having k false positives, for $k = 0, \ldots, n$, is $\binom{n}{k} n^{-k}(1 - 1/n)^{n-k} \approx e^{-1}/k!$, where n is either 2^{64} or 2^{96}.

Starting from a single state of the pool at time t, let us denote by d_i the number of false positives at time $t - i$. Note that for every i there exists, in addition to the false positives, an additional candidate which is the true value of the pool at time $t - i$.

In time t we have one good candidate and no false positives ($d_0 = 0$). Now,

$$E(d_1) = \sum_{k=1}^{n} ke^{-1}/k! = e^{-1} \sum_{k=0}^{n-1} 1/k! = e^{-1} \cdot e = 1.$$

It also holds that $E(d_i|d_{i-1} = 1) = E(d_1) = 1$. As a result, $E(d_i|d_{i-1} = c) = (c + 1)E(d_i|d_{i-1} = 1) = c + 1$. (We multiply $E(d_i|d_{i-1} = 1)$ by $c + 1$ since we obtain false positives at time $t - i$ from the c false positives at time $t - i + 1$ and in addition from the true value of $\mathrm{pool_{t-i+1}}$.)

A martingale is a sequence of random variables $X_0, X_1, X_2, \ldots$ which satisfies the relation

$$E(X_i|X_{i-1}, \ldots, X_0) = X_{i-1}.$$

It is known that for a martingale $E(X_i) = E(X_0)$, and there are known tail bounds on the divergence from this expectation (see [17] for details).

Let us define the sequence $z_i = d_i - i$ (the deviation of d_i from the value i). The sequence $\{d_i\}$ is not a martingale but the sequence $\{z_i\}$ is, since $E(z_i|z_{i-1} = c) = E(d_i|d_{i-1} = c + i - 1) - i = c + i - 1 + 1 - i = c = z_{i-1}$. We therefore get that $E(z_i) = E(z_0) = 0$ and $E(d_i) = i$.

It is now possible to apply the Kolmogorov-Doob inequality (see [17]), which states that $\Pr(\max(z_i) > b) < E(z_0)/b = 1/b$ (for the purpose of using this inequality we define $z_i = d_i - i + 1$ and therefore $z_0 = 1$). As a corollary we can obtain, for example, that the probability that d_i is greater than $i + 100$ is at most $1/100$.