The Linux Pseudorandom Number Generator Revisited

Patrick Lacharme, Andrea Rock, Vincent Strubel, Marion Videau

The Linux Pseudorandom Number Generator Revisited

Patrick Lacharme *
Andrea Röck †
Vincent Strubel ‡
Marion Videau §

Abstract

The Linux pseudorandom number generator (PRNG) is a PRNG with entropy inputs which is widely used in many security related applications and protocols. This PRNG is written as an open source code which is subject to regular changes. It was last analyzed in the work of Gutterman et al. in 2006 [GPR06] but since then no new analysis has been made available, while in the meantime several changes have been applied to the code, among others, to counter the attacks presented in [GPR06]. Our work describes the Linux PRNG of kernel versions 2.6.30.7 and upwards. We detail the PRNG architecture in the Linux system and provide its first accurate mathematical description and a precise analysis of the building blocks, including entropy estimation and extraction. Subsequently, we give a security analysis including the feasibility of cryptographic attacks and an empirical test of the entropy estimator. Finally, we underline some important changes to the previous versions and their consequences.

1 Introduction

The security of many protocols is based on the impossibility for an attacker to guess random data, such as session keys for cryptosystems or nonces for cryptographic protocols. The frequency and the amount of required random data can differ greatly with the application. Therefore, random data generation should take into account the fact that the user can request either high quality random data or a great amount of pseudorandom data. There are several types of PRNGs: *non-cryptographic deterministic PRNGs* which should not be used for security applications, *cryptographically secure PRNGs* (CSPRNGs) which are deterministic algorithms with outputs that are unpredictable to an outsider without knowledge of the generator's internal states and *PRNGs with entropy inputs* (see e.g. [MvOV96, Ch. 5] for classification and descriptions). The Linux PRNG falls into this last category.

A pseudorandom number generator with entropy inputs produces bits non-deterministically as the internal state is frequently refreshed with unpredictable data from one or several external entropy sources. It is typically made up of (1) several physical sources of randomness called entropy sources, (2) a harvesting mechanism to accumulate the entropy from these sources into

*Ensicaen - UCBN - CNRS, GREYC UMR 6072, Caen, F-14000, France
†Cryptolog, Paris, F-75011, France. The work of the author was realized while she was at Aalto University, Finland, has been funded by the Academy of Finland under project 122736 and was partly supported by the European Commission through the ICT program under contract ICT-2007-216676 ECRYPT II.
‡Agence nationale de la sécurité des systèmes d'information (ANSSI), Paris, F-75007, France
§Université de Lorraine - CNRS - INRIA, LORIA UMR 7503, Vandœuvre-lès-Nancy, F-54500, France. The work of the author was realized while she was at Agence nationale de la sécurité des systèmes d'information and was partially supported by the French Agence Nationale de la Recherche under Contract ANR-06-SETI-013-RAPIDE.

the internal state, and (3) a post-processing procedure. The latter frequently uses a CSPRNG to generate outputs and to update the internal state.

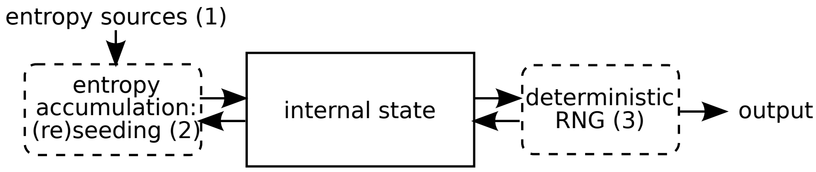

Figure 1: Model of PRNG with entropy inputs.

Previous work. PRNGs with entropy inputs are crucial elements for the security of systems and many papers on the topic are available. Gutmann [Gut98] proposes an analysis of many PRNGs, with a comprehensive guide to designing and implementing them. PRNG designs are also discussed by Kelsey et al. for the analysis of Yarrow [KSF99] and by Fergusson and Schneier for the analysis of Fortuna [FS03, Ch. 10]. Other examples are the Intel PRNG [JK99], Havege, by Seznec and Sendrier [SS03], and the Windows PRNG. The latter has been analyzed by Dorrendorf et al. in e.g. [DGP09], where a flaw in the forward security is reported. In [BH05], Barak and Halevi discuss a theoretical model for a PRNG with entropy inputs and compare it to the Linux PRNG. General recommendations on the subject are given in RFC 4086 [ESC05] and a detailed guideline is proposed by the NIST [BK07].
Previous analysis of the Linux PRNG. A detailed analysis of the Linux PRNG was done by Gutterman et al. in 2006 [GPR06], based on kernel version 2.6.10, released in 2004. The authors proposed a forward attack, which enables an attacker with knowledge of the internal state to recover previous states with a complexity of 2^{64} or 2^{96}, depending on the attack. In addition, they presented a denial of service attack on the blocking variant of the PRNG. Since then, several changes have been made in the source code of the Linux PRNG, some aiming at preventing these attacks. There has been no published analysis of the newer version.

Our contribution. This document details the Linux PRNG for kernel versions starting from 2.6.30.7 [1]. The architecture of the generator is presented in Section 2. In Section 3, we discuss the mathematical details of the building blocks used in the generator and their properties, and suggest some improvements. We examine the functions used to mix data into a pool (mixing function) or to generate data from a pool (output function), as well as the entropy estimator which is a crucial element for `/dev/random`. Section 4 presents the security requirements and the security analysis of the PRNG inculding empricial tests of the entropy estimator. Finally, changes from the version analyzed in [GPR06] are outlined in Section 5.

2 Architecture

The Linux PRNG is part of the Linux kernel since 1994. The original version was written by Ts'o, and later modified by Mackall [MT09]. The generator, apart from the entropy input hooks inserted into e.g. drivers, represents about 1700 lines of C code in a single source file, `drivers/char/random.c`.

[1] The changes made to stable versions of the kernel from 2.6.30.7 up to and including 3.1.10 have no impact on our analysis. Version 3.2.0 of the kernel introduced the use of the "RDRAND" instruction, to extract random bytes from a hardware RNG included in the latest Intel CPU chips (Ivy Bridge architecture). This mode of operation is not covered in our analysis, which however remains valid for a 3.2.* kernel running on any other CPU, including all other Intel chips.

2.1 General Structure

Unlike others PRNGs, the internal state of the Linux PRNG is composed of three pools, namely the *input pool*, the *blocking pool* and the *nonblocking pool*, according to the source code. In this paper the two last pools are also referred as output pools, depending on the context. The PRNG relies on external entropy sources. Entropy samples are collected from system events inside the kernel, asynchronously and independently from output generation. These inputs are then accumulated into the *input pool*.

The generator is designed to perform the collection of entropy inputs as efficiently as possible. Therefore it uses a linear mixing function instead of a more usual hash function. The security of the generator strongly relies on the cryptographic primitive Sha-1, which is used for *output generation* and *entropy transfers* between the input pool and the output pools. The design of the generator is illustrated by Figure 2. The size of the input pool is 128 32-bit words (4096 bits)

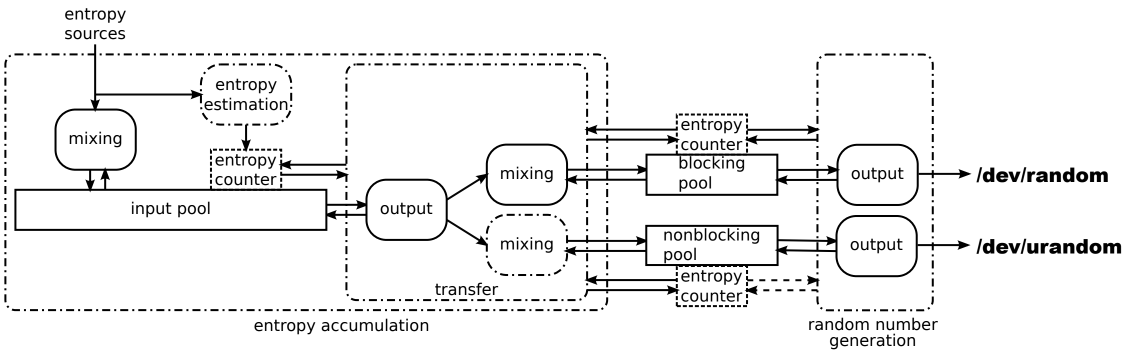

Figure 2: General structure of the Linux PRNG.

and the size of each output pool is 32 32-bit words (1024 bits). Each pool has its own *entropy counter*, which is decremented when random bits are extracted from the pool and incremented when new inputs are collected and mixed into the pool. Entropy is only transfered to an output pool when a corresponding number of bytes need to be extracted from it. Thus, the entropy counters of the output pools will generally remain close to zero, except in transitional states during the extraction. Therefore the only significant entropy counter is that of the input pool.

User space provides `/dev/random` and `/dev/urandom` which are two different character device interfaces to read random outputs. The `/dev/random` device reads from the blocking pool and limits the number of generated bits according to the estimation of the entropy available in the PRNG. Reading from this device is blocked when the PRNG does not have enough entropy, and resumed when enough new entropy samples have been mixed into the input pool. It is intended for user space applications in which a small number of high quality random bits is needed. The `/dev/urandom` device reads from the nonblocking pool and generates as many bits as the user asks for without blocking. It is meant for the fast generation of large amounts of random data. Writing data to either one of these two devices mixes the data in both the blocking and nonblocking pools, without changing their entropy counters. In addition to these two user space interfaces, the Linux PRNG provides a single kernel interface, through the function `get_random_bytes()`, which allows other kernel components to read random bytes from the nonblocking pool. There is no kernel interface to the blocking pool, which is reserved for user space applications.

2.2 Entropy Inputs

Entropy inputs are injected into the generator for initialization and through the updating mechanism. This provides the backbone of the security of the PRNG. The Linux PRNG is intended to be usable independently of any specific hardware. Therefore, it cannot rely on physical nondeterministic phenomena generally used in random generation, which require additional hardware. In fact, even though the Linux kernel includes drivers for a number of hardware RNGs, the outputs from these generators, when present, are made available only to user space, through a specific character device (`hwrng`), and are not mixed into the Linux PRNG. It is up to user space applications (e.g. an entropy gathering daemon) to collect these outputs and feed them into the PRNG if needed.

The Linux PRNG processes events from different entropy sources, namely user inputs (such as keyboard and mouse movements), disk timings and interrupt timings. To avoid events from very regular interrupts, each device driver can define if its interrupts are suitable as entropy inputs, by adding the `IRQF_SAMPLE_RANDOM` flag to the corresponding handler. This generic method for adding new sources of interrupt entropy was typically used by most network card drivers and some USB device drivers. It has however been scheduled for removal since 2009 (as stated in the `feature-removal-schedule.txt` file within the kernel source tree), due to several misuses. It should in time be replaced by more precisely defined sources of entropy. Unfortunately the gradual removal of this flag from device interrupts has in the meantime left input events and disk timings as the only two reliable sources of entropy in recent kernel versions. This might be a security problem in some use cases, where user inputs are nonexistent and disk accesses easily predictable.

For each entropy event fed into the PRNG, three 32-bit values are considered: the *num* value, which is specific to the type of event [2], the current CPU *cycle* count and the *jiffies* count at the time the event is mixed into the pool. The jiffies count corresponds to the internal kernel counter of timer interrupts since the last kernel boot. The frequency of the timer interrupt is defined at build time by the `HZ` parameter in the kernel source, and generally ranges between 100 to 1000 ticks per second [CRKH05, Ch. 7]. These values provide much less than 32 bits of entropy each. As shown in [GPR06, Table 1], the maximal entropy of the 32-bit `num` value is 8 bits for keyboard events, 12 bits for mouse events, 3 bits for hard drive events, and 4 bits for interrupts. As can be seen in Table 1 in Section 4.2, the average empirical entropy of `num` for user input events (keyboard and mouse movements) is even less. The empirical entropy of the jiffies and the cycle counts for user inputs in our tests was only around 3.4 and 14.8 bits, respectively. However, the generator never assumes that the events injected into the input pool provide maximal entropy. It tries to estimate the entropy in a pessimistic way so as to not overestimate the amount it collects.

2.3 Entropy Accumulation

Entropy samples are added to the input pool using the mixing function described in Section 3.1. The entropy counter of the input pool is incremented according to the estimated entropy of the mixed data. The same mixing function is also used when transferring data from the input pool to one of the output pools, when the latter requires more entropy for output generation. In that case, the algorithm assumes full entropy of the data, and the entropy counter of the output pool is incremented by the exact number of transferred bits.

[2] For example, in the case of a keyboard event, *num* is derived from the keycode number.

2.3.1 Data Injection

Definition 1. *Let* $\mathbf{X}$ *be a* n*-bit random variable describing the internal state and* $\mathbf{I}$ *an* m*-bit random variable corresponding to the input sample of the entropy source. Let* f *be a function from* $\{0,1\}^n \times \{0,1\}^m$ *to* $\{0,1\}^n$*, and* H *Shannon's entropy function (see Definition 3, Appendix A). The function* f *is a* mixing function *if and only if*

$$H(f(\mathbf{X},\mathbf{I})) \geq H(\mathbf{X}) \text{ and } H(f(\mathbf{X},\mathbf{I})) \geq H(\mathbf{I}).$$

Definition 1 means that a mixing function is never counterproductive. It does not mean that the entropy is mixed into the internal state in an optimal way or that no entropy gets lost. However, it guarantees that if an attacker has no knowledge of the state $\mathbf{X}$ but has complete control of the input $\mathbf{I}$, he will gain no additional information on the new state after the execution of the mixing function. Conversely, if the attacker knows the internal state but not the new input, the whole entropy of $\mathbf{I}$ can be gained. The mixing function of the Linux PRNG verifies this definition, as we show in Section 3.1.

2.3.2 Entropy Estimation

The estimation of available entropy is crucial for `/dev/random`. It must be fast and provide an accurate estimation of whether the corresponding pool contains enough entropy to generate unpredictable output data. It is important not to overestimate the entropy provided by input sources.

Entropy estimation is based on a few reasonable assumptions. It is assumed that most of the entropy of the input samples is contained in their timings. Both the cycle and jiffies counts can be seen as a measure of timing, however the jiffies count has a much coarser granularity. The Linux PRNG bases its entropy estimation on the jiffies count only, which leads to a pessimistic estimation. Adding additional values, which are not used for the estimation, can only increase the entropy. Even adding completely known input cannot decrease the uncertainty of the already collected data as showed in Lemma 1. The input samples come from different sources, which can be assumed to be independent. Entropy can therefore be estimated separately for each source and summed up in the end. The estimator considers several different sources: user input, interrupts, and disk I/O. Each interrupt request (IRQ) number is seen as a separate source. The estimator keeps track of the jiffies count of each source separately. The values of the jiffies count are always increasing, except in the rare case of an overflow. The entropy is estimated from the jiffies difference between two events.

An entropy estimator for entropy sources has to deal with several constraints. The method of estimation used in this PRNG is detailed and discussed in Section 3.2. The conditions listed below apply to the Linux entropy estimator:

Condition 1. *Unknown and non-uniform distribution*: The distribution of the input samples can vary a lot depending on the situation in which the PRNG is used. Therefore, no assumption can be made on the input. In Figure 7, we present a part of the empirical distribution of jiffies differences measured from a sample of user inputs (as described in Section 4.2). The distribution is clearly non-uniform.

Condition 2. *Unknown correlation*: It is very likely that there are some correlations between the input samples. This is the case for instance when the user is typing some text. However, it may be difficult to capture them precisely.

Condition 3. *Large sample space*: The jiffies differences are measured in 32 or 64-bit values (depending on the size of the C `long` type) which can be arbitrarily high. It creates a sample space of size 2^{32} or 2^{64}. This makes it very hard to keep track of the occurrences of all values.

Condition 4. *Limited time*: The estimation happens after interrupts so it cannot take much computation time or memory space.
Condition 5. *Estimation at runtime*: The amount of uncertainty must be estimated for each input sample. Therefore an estimator which waits for multiple events before estimating the entropy from the empirical frequencies cannot be used.
Condition 6. *Unknown knowledge of the attacker*: As for any other estimator, there is no information about the knowledge of a potential attacker.

2.4 Output Generation

The random data generation is done in blocks of 10 output bytes. For each output block, 20 bytes, produced during the process, are injected and mixed back into the source pool to update it. If the number of requested bytes is not a multiple of 10, the last block is truncated to the length of the missing bytes. When k bytes need to be generated, the generator first checks whether there is enough entropy in the current output pool according to its entropy counter. If this is the case, k output bytes are generated from this pool and the entropy counter is decreased by k bytes. Otherwise, if there is not enough entropy in the output pool, the PRNG requests a transfer of k bytes (at least 8 and at most 128) of entropy from the input pool into the output pool. The actual output function used for output generation and transfer is precisely described in Section 3.3.

The transfer is done by first producing k' bytes from the input pool using the output function, then injecting those k' bytes into the output pool with the mixing function. The k' value depends on the entropy count h_{I} (in bits) of the input pool and on the requesting pool. If the request comes from the blocking pool, then $k' = \min(\lfloor h_{\mathrm{I}}/8 \rfloor, k)$, whereas if it comes from the nonblocking pool, $k' = \min(\lfloor h_{\mathrm{I}}/8 \rfloor - 16, k)$. This means that the input pool does not generate more bytes than its entropy counter allows. Moreover, if the request comes from the nonblocking pool it leaves at least 16 bytes of entropy in the input pool. If $k' < 8$, no bytes are transferred to avoid frequent work for very small amounts of data[3].

After the transfer, the entropy counters of the input and output pools are respectively reduced and increased by $8k'$ bits. Due to this transfer policy, the entropy counters of the output pools remain most of the time close to zero between two output requests, since only as many bytes are transferred as are needed for the output. No output is generated from the output pool before all k' bytes have been injected. During the injection, the output pool is shifted k' times by the mixing function. For every 10 bytes generated from the output pool, 20 bytes are mixed back and the output pool gets shifted 20 times. Thus, to produce k bytes, the output pool is shifted at least $2k$ times, when no transfer of entropy data is necessary, and at most $2k + k'$ times, if k' bytes are transferred from the input pool.

Let h_{O} denote the entropy counter of the output pool after the entropy transfer. In the case of `/dev/random`, if $h_{\mathrm{O}} < 8k$, and there are less than 8 bytes of estimated entropy in the input pool, output generation stops after $\lfloor h_{\mathrm{O}}/8 \rfloor$ bytes, and only resumes when enough entropy has been mixed into the input pool for a transfer to occur. In contrast, `/dev/urandom` continues to output data until all k bytes have been produced, regardless of whether the input pool had enough entropy to satisfy all transfer requests.

[3]This minimum transferred size of 8 bytes is the default value. It can be globally modified by privileged user space applications through the `kernel.random.read_wakeup_threshold sysctl` variable (in bits). Such modifications also affect the reserved entropy left in the input pool by transfers to the nonblocking pool, which is twice the minimum transferred size.

2.5 Initialization

The Linux boot process does not provide much entropy in the different sources available to the PRNG. There is usually little to no user input and network events at this stage, and disk activity at startup is very deterministic. Therefore, the designer of the Linux PRNG recommends a script which, at shutdown, generates data from `/dev/urandom` and saves it in a file, and at startup, writes the saved data to `/dev/urandom`. This mixes the same data into the blocking and nonblocking pools without increasing their entropy counters.

Such a script is provided in the default installation of most Linux distributions. In situations where this procedure is not possible, for example in Live CD systems, the nonblocking random number generator should be used with caution directly after the boot process since it might not contain enough entropy.

3 Building Blocks

In this section, we give a detailed mathematical analysis of the building blocks of the Linux PRNG.

3.1 The Mixing Function

This procedure mixes one byte at a time by first extending[4] it to a 32-bit word, then rotating it by a changing factor and finally mixing it in the pool by using a linear shift register. It is designed so that it can diffuse entropy into the pool and no entropy gets lost. The function has not changed since 2006 and is presented in Figure 3.

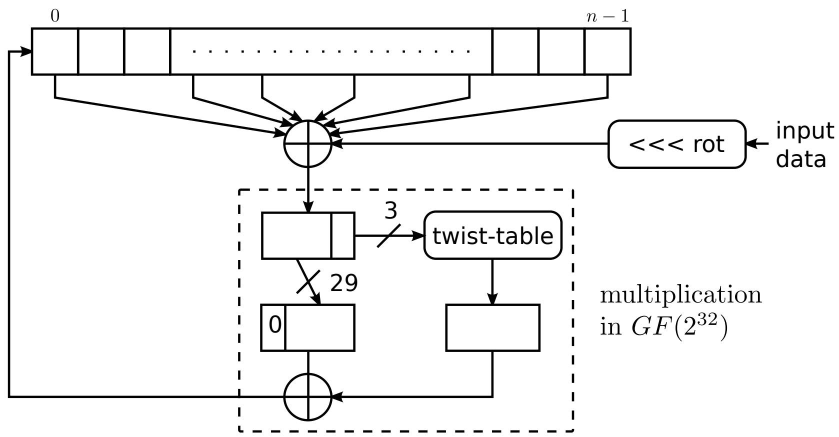

Figure 3: The mixing function.

The mixing function must properly process entropy inputs (as will be discussed in Section 3.1.2) and should also handle the case where there is no entropy input. In this latter case one expects the successive internal states to generate a sequence with maximal period. We study this case in Section 3.1.1. Most notably, we show that the mixing function of the Linux PRNG is indeed a *mixing function* according to Definition 1 and that the mixing function without input *does not* generate a sequence of internal states with a maximal period. Moreover, we show that the

[4]This 32-bit extension is either signed or unsigned, depending on the default signedness of the *char* type on the considered CPU architecture. The difference is assumed not to be significant.

theoretical background presented in the comments of the source code is somehow out of date, not completely relevant and furthermore faultily understood.

3.1.1 Analysis Without Input

When the input is set to zero, the mixing function is equivalent to an LFSR over $GF(2^{32})$ with feedback polynomial $Q(X) = \alpha^3(P(X) - 1) + 1$, where α is the primitive element of $GF(2^{32})$ corresponding to X defined by the CRC-32-IEEE 802.3 polynomial, and $P(X)$ depends on the size of the pool:

input pool: $P(X) = X^{128} + X^{103} + X^{76} + X^{51} + X^{25} + X + 1$
output pool: $P(X) = X^{32} + X^{26} + X^{20} + X^{14} + X^7 + X + 1$.
The multiplication by α^3 is done by a lookup table, called *twist-table* in the source code.

The actual system slightly differs from what is stated in the comments of the source code. First, the design of the mixing function is claimed to rely on a Twisted Generalized Feedback Shift Register (TGFSR) as defined in [MK92]. However, TGFSRs are LFSRs on binary words with a trinomial feedback polynomial whereas the mixing function uses a heptanomial. The case of general polynomials on finite fields is treated in standard literature, such as [LN97]. Moreover, the maximal period, that is the primitivity of the feedback polynomial, seems to have been ill understood, as the comments mention the primitivity for polynomials on $GF(2)$, whereas the primitivity must be checked on $GF(2^{32})$. This confusion is also repeated in [GPR06, Definition 2.2].

Finally, the polynomial $Q(X) = \alpha^3(P(X) - 1) + 1$ is not primitive over $GF(2^{32})$, nor is it even irreducible. Thus, the resulting LFSR does not achieve maximal period. The period is less than $2^{92*32} - 1$, rather than the maximal value of $2^{128*32} - 1$, for the input pool, and less than $2^{26*32} - 1$ instead of $2^{32*32} - 1$ for the output pool. We do not believe these reduced periods can lead to practical attacks. However, $Q(X)$ can be made irreducible by changing just one feedback position:

input pool: $P(X) = X^{128} + \mathbf{X^{104}} + X^{76} + X^{51} + X^{25} + X + 1$
output pool: $P(X) = X^{32} + X^{26} + \mathbf{X^{19}} + X^{14} + X^7 + X + 1$.

These modified polynomials have periods of $(2^{128*32} - 1)/3$ and $(2^{32*32} - 1)/3$, respectively. A primitive polynomial can be easily achieved by using $\alpha^i(P(X) - 1) + 1$ with $\gcd(i, 2^{32} - 1) = 1$, for example for $i = 1, 2, 4, 7, \ldots$, and an adequate polynomial $P(X)$. This would change the size of the twist-table to 2^i elements. For instance, $\alpha^{\mathbf{2}}(X^{32} + X^{26} + \mathbf{X^{23}} + X^{14} + X^7 + X) + 1$ is primitive. All these computations can be made using computational algebra systems like MAGMA [BCP97].

3.1.2 Analysis With Input

The mixing function can be rearranged and presented by means of two linear functions $L_1 : \{0,1\}^8 \to \{0,1\}^{32}$ and $L_2 : (\{0,1\}^{32})^5 \to \{0,1\}^{32}$ as in Figure 4. The L_1 function takes the 8-bit input y, extends it to 32 bits, rotates it and applies the multiplication in $GF(2^{32})$ by means of the twist-table. The $L_2(x_0, x_{i_1}, x_{i_2}, x_{i_3}, x_{i_4}, x_{i_5})$ function represents the feedback function.
We believe the comments in the source code, which refer to universal hash functions, are irrelevant to assess the security of the mixing function's design. Therefore, a careful analysis is required to make sure that the function properly processes entropy inputs in regard to the control or knowledge an attacker may have of either the inputs or the internal state, i.e. that it verifies Definition 1.

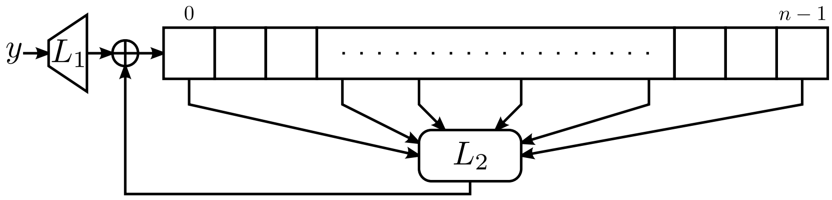

Figure 4: The mixing function.

Lemma 1. *Let* $X = (X_0, \dots, X_{n-1})$ *be a random variable representing the internal state, where each* X_i *represents a 32-bit word and* $n = 32$ *or* $n = 128$*, depending on the pool, and let* $(0, i_1, i_2, i_3, i_4, n-1)$ *be the indices of the words used in the feedback function. Let* $X_{[i,j]}$ *denote the sub-part* $X_i, \dots, X_j$ *of the state* X *and let* Y *be a random variable representing the 8-bit input. We assume that* X *and* Y *are statistically independent. The mixing function of the Linux PRNG*

$$\begin{array}{lcl} f(Y, X) & = & (\tilde{X}_0, \dots, \tilde{X}_{n-1}) \\ \tilde{X}_0 & = & L_1(Y) \oplus L_2(X_0, X_{i_1}, X_{i_2}, X_{i_3}, X_{i_4}, X_{n-1}) \\ \tilde{X}_i & = & X_{i-1}, \textit{ for } 1 \le i \le n-1 \end{array} \tag{1}$$

is a mixing function *according to Definition 1. Moreover, we can show that*

$$H(f(Y, X)) \ge \max\Big(H(Y), H(X_{n-1}|X_{[0,n-2]})\Big) + H(X_{[0,n-2]}).$$

Proof. This proof relies on some basic properties of the entropy function. For a detailed discussion of this topic we refer to standard literature such as [CT06]. The main entropy can be written as joined and conditional entropy:

$$\begin{array}{rcl} H(f(Y, X)) & = & H(L_1(Y) \oplus L_2(X), X_{[0,n-2]}) \\ & = & H(L_1(Y) \oplus L_2(X)|X_{[0,n-2]}) + H(X_{[0,n-2]}). \end{array}$$

We use that for any *injective* function g and any discrete random variable Z: $H(g(Z)) = H(Z)$. For a fixed value of $(X_{[0,n-2]}, X_{n-1}) = x$, the function $L_1(\cdot) \oplus L_2(x)$ is injective and thus $H(L_1(Y) \oplus L_2(X)|X) = H(Y|X)$. Since the state X and input Y are independently distributed we have $H(Y|X) = H(Y)$ and we can write:

$$H(L_1(Y) \oplus L_2(X)|X_{[0,n-2]}, X_{n-1}) = H(Y). \tag{2}$$

In a similar way, for fixed values $X_{[0,n-2]} = x_{[0,n-2]}$ and $Y = y$, the function $L_1(y) \oplus L_2(x_{[0,n-2]}, \cdot)$ is injective and X and Y are independent thus:

$$H(L_1(Y) \oplus L_2(X)|Y, X_{[0,n-2]}) = H(X_{n-1}|X_{[0,n-2]}). \tag{3}$$

For any random variables Z_1, Z_2 it holds that $H(Z_1) \ge H(Z_1|Z_2)$. Thus from 2 and 3, it follows

$$\begin{array}{rcl} H(f(Y, X)) & \ge & H(Y) + H(X_{[0,n-2]}) \text{ and} \\ H(f(Y, X)) & \ge & H(X_{n-1}|X_{[0,n-2]}) + H(X_{[0,n-2]}) = H(X), \end{array}$$

which concludes our proof. □

Remark 1. *Lemma 1 is true for any* injective *function* L_1 *and any function* L_2 *that is* injective in X_{n-1}.

3.2 The Entropy Estimator

In the following we denote random variables $\mathbf{X}, \mathbf{T}, \boldsymbol{\Delta}, \ldots$ by capital boldface letters and their corresponding sample spaces by $\mathcal{X}, \mathcal{T}, \mathcal{D}$. The realizations $x \in \mathcal{X}, t \in \mathcal{T}, \delta \in \mathcal{D}, \ldots$ of random variables are marked by small letters. The probability distribution of $\mathbf{X}$ is defined by $p^{\mathbf{X}} = \{p^{\mathbf{X}}(\eta)\}_{\eta \in \mathcal{X}}$ where $p^{\mathbf{X}}(\eta) = Pr[\mathbf{X} = \eta]$ is the probability of $\mathbf{X}$ being $\eta \in \mathcal{X}$. We may omit $\mathbf{X}$ in the notation when its meaning is clear from the context. Values based on a sequence of empirical data $x_0, x_1, \ldots, x_n$ are written with a hat, like $\hat{p}_\eta = \#\{0 \leq i \leq n : x_i = \eta\}/n$ for the empirical frequency of η or $\hat{H} = -\sum_{\eta \in \mathcal{X}} \hat{p}_\eta \log_2 \hat{p}_\eta$ for the empirical entropy.

3.2.1 Implementation of the Estimator

In this section we give a detailed description of the actual estimator used in the Linux PRNG. Its practical application is discussed in Section 4.2. Since the estimation is done separately for each entropy source, we analyze the case where the whole data comes from a single source.

Let $\mathbf{T}_0, \mathbf{T}_1, \ldots$ denote the input sequence to the estimator. The sequence represents the jiffies counts of the events, and is thus an increasing sequence (except for very rare counter overflows). Since the estimation of the entropy should not depend on the time elapsed since the system was booted (beginning of the jiffies count), only the sequence of time differences $\boldsymbol{\Delta}_i^{[1]} = |\mathbf{T}_i - \mathbf{T}_{i-1}|$ are considered. Counter overflows are handled transparently by considering the absolute value of the differences. We now consider the sequence of random variables $\boldsymbol{\Delta}_1^{[1]}, \boldsymbol{\Delta}_2^{[1]}, \ldots$ assuming that they are identically (but not necessarily independently) distributed, based on the fact that they come from a single source (even if all "user inputs" count as a single source). We denote by $\mathcal{D}$ the sample space of the $\boldsymbol{\Delta}_i^{[1]}$'s with a size of $D = |\mathcal{D}| \gg 2$. Thus, $\delta_i^{[1]}$ corresponds to the realization of the jiffies difference at time i for the considered source. For the estimator we define the three following random variables for $i \geq 3$:

$$\begin{aligned}
\boldsymbol{\Delta}_i^{[2]} &= \boldsymbol{\Delta}_i^{[1]} - \boldsymbol{\Delta}_{i-1}^{[1]} \\
\boldsymbol{\Delta}_i^{[3]} &= \boldsymbol{\Delta}_i^{[2]} - \boldsymbol{\Delta}_{i-1}^{[2]} = \boldsymbol{\Delta}_i^{[1]} - 2\boldsymbol{\Delta}_{i-1}^{[1]} + \boldsymbol{\Delta}_{i-2}^{[1]} \\
\boldsymbol{\Delta}_i &= \min\left(|\boldsymbol{\Delta}_i^{[1]}|, |\boldsymbol{\Delta}_i^{[2]}|, |\boldsymbol{\Delta}_i^{[3]}|\right).
\end{aligned}$$

We also define a logarithm function that is bounded by a maximal output of 11 and returns integer values:

$$LOG_2(m) = \begin{cases} 0 \text{ if } m < 2 \\ 11 \text{ if } m \geq 2^{12} \\ \lfloor \log_2(m) \rfloor \text{ otherwise} \end{cases}$$

Then, for a specific outcome $\delta_1^{[1]}, \delta_2^{[1]}, \ldots$ the estimation of the entropy received at time i, is defined by:

$$\hat{H}_i^{[3]} = \hat{H}^{[3]}\left(\delta_i^{[1]}, \delta_{i-1}^{[1]}, \delta_{i-2}^{[1]}\right) = LOG_2(\delta_i). \tag{4}$$

If not stated otherwise, we will assume that the $\boldsymbol{\Delta}_i^{[1]}$ are independently distributed. Without loss of generality we further assume that $\mathcal{D} = \{0, 1, \ldots, D-1\}$. Then $p(\eta) = Pr[\delta = \eta]$ is the probability of the difference being $0 \leq \eta < D$. To compute the value of $\hat{H}_i^{[3]}$, we have to know t_i, t_{i-1}, $\delta_{i-1}^{[1]}$, $\delta_{i-1}^{[2]}$. Thus, for each source the estimator stores three values t_{i-1}, $\delta_{i-1}^{[1]}$, $\delta_{i-1}^{[2]}$ between two events, which only requires a very small amount of memory and computations.

One basic property of any entropy definition is that it is invariant under a permutation of the sample space. This means that if we define for any permutation $\pi : \mathcal{X} \rightarrow \mathcal{X}$ the distribution q with $q_\eta = p_{\pi(\eta)}$, $H(p) = H(q)$ always holds. This is not true for this estimator, since it uses the value of a given element and not its probability. Thus there cannot be a theoretical link between the estimator and the entropy which holds for all distributions.

3.3 The Output Function

The output function is the only non-linear operation of the Linux PRNG. It is used in two cases: when data is transferred from the input pool into one of the output pools and when output data is generated from one of the output pools.

This function uses the Sha-1 hash function in two steps and is detailed in Figure 5. In the first step, the hash of the whole pool is computed and the digest mixed back to update the pool. In the second step, the final output is generated. These two steps can be named the feedback phase and the extraction phase, respectively.

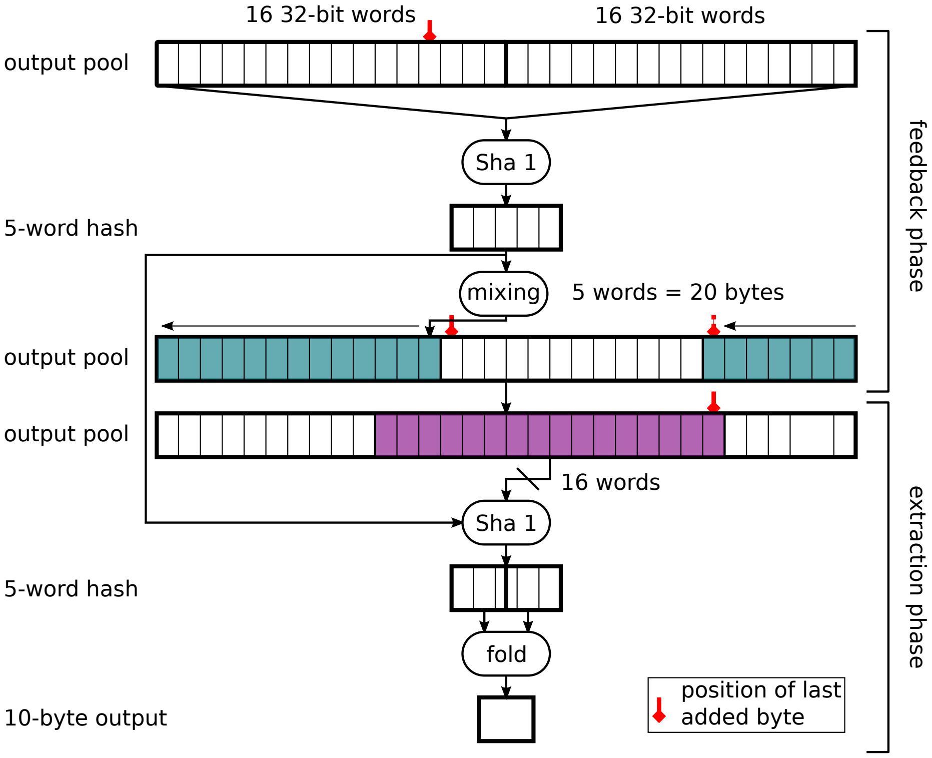

Figure 5: The output function of the Linux PRNG.

First step: the feedback phase. All the bytes of the pool are fed into the Sha-1 hash function, to produce a 5-word (20-byte) hash. These 20 bytes are mixed back into the pool by using the mixing function. Consequently, the pool is shifted 20 times for each feedback phase. This affects 20 consecutive words (640 bits) of the pool.

Second step: the extraction phase. Once mixed with the pool content, the 5 words computed in the first step are used as an initial value or chaining value when hashing another 16 words from the pool. These 16 words overlap with the last word changed by the feedback data. In the case of an output pool (pool length = 32 words), they also overlap with the first 3 changed words. The 20 bytes of output from this second hash are folded in half to compute the 10 bytes to be extracted: if $w_{[m\ldots n]}$ denotes the bits $m, \ldots, n$ of the word w, the folding operation of the five words w_0, w_1, w_2, w_3, w_4 is done by $w_0 \oplus w_3, w_1 \oplus w_4, w_{2_{[0\ldots15]}} \oplus w_{2_{[16\ldots31]}}$.

Finally, the estimated entropy counter of the affected pool is decremented by the number of generated bytes.

3.3.1 Entropy Extraction

In this section we analyze the entropy of the generated data. Assuming that the pool contains k bits of *Rényi entropy* of order 2 (see Definition 3, Appendix A), a 2-universal hash function [WC81] can be used to extract almost uniform data. This property is called *privacy amplification* [BBR88].

Definition 2. *A* 2-universal hash function *is a set $\mathcal{G}$ of functions $\mathcal{X} \to \mathcal{Y}$ such that for all distinct elements x, x' there are at most $|\mathcal{G}|/|\mathcal{Y}|$ functions $g \in \mathcal{G}$ such that $g(x) = g(x')$.*

Theorem 1 (Privacy Amplification [BBR88])**.** *Let $\mathbf{X}$ be a random variable over the alphabet $\mathcal{X}$ with probability distribution $p^{\mathbf{X}}$ with Rényi entropy $H_2(\mathbf{X})$, let $\mathbf{G}$ be a random variable corresponding to the random choice of the universal class of hash functions $\mathcal{X} \to \{0,1\}^r$ and let $\mathbf{Y} = \mathbf{G}(\mathbf{X})$. Then*

$$H(\mathbf{Y}|\mathbf{G}) \geq H_2(\mathbf{Y}|\mathbf{G}) \geq r - \frac{2^{r-H_2(\mathbf{X})}}{\ln(2)}.$$

This means that if $H_2(\mathbf{X}) \geq r$ and $\mathbf{G}$ is uniformly distributed, then the entropy in the output is close to r bits, even if the specific function $\mathbf{G} = g$ is known.

In the case of the Linux PRNG the hash function $h : \mathcal{X} \to \mathcal{Y}$ is fixed, with $\mathcal{X} = \{0,1\}^n$, $\mathcal{Y} = \{0,1\}^r$, and $r < n$. However, the theorem still applies with two additional assumptions. First, let us assume that each element $y \in \mathcal{Y}$ has a preimage of size $\#\{x|h(x) = y\} = |\mathcal{X}|/|\mathcal{Y}|$. This corresponds to the definition of a *regular* function in [GKL93]. If this assumption was far from being true, a birthday attack would lead to a collision with a complexity of less than $\mathcal{O}(2^{r/2})$, which should not be the case for a cryptographic hash function. Secondly, we assume that the attacker knows the probability distribution of $\mathbf{X}$, and thus its entropy, but cannot influence it. Let Π be the set of all permutations $\pi : \mathcal{X} \to \mathcal{X}$. For a given probability distribution $p^{\mathbf{X}} = \{p^{\mathbf{X}}(\eta)\}_{\mathcal{X}}$ with entropy $H(\mathbf{X})$, all distributions $q_\pi^{\mathbf{X}} = \{p^{\mathbf{X}}(\pi(\eta))\}_{\eta\in\mathcal{X}}$ for $\pi \in \Pi$ can appear with the same probability. We assume that the attacker knows but cannot choose which π has been used.

With these two assumptions, we can express the class of universal hash functions as $\mathcal{G} = \{h \circ \pi\}_{\pi\in\Pi}$. We have $|\mathcal{G}| = |\mathcal{X}|!$ and for any pair $(x_1, x_2) \in \mathcal{X}^2$ the number of functions $g \in \mathcal{G}$ such that $g(x_1) = g(x_2)$ is

$$|\mathcal{Y}|\frac{|\mathcal{X}|}{|\mathcal{Y}|}\left(\frac{|\mathcal{X}|}{|\mathcal{Y}|} - 1\right)(|\mathcal{X}| - 2)! = \frac{|\mathcal{X}|!}{|\mathcal{Y}|}\frac{|\mathcal{X}| - |\mathcal{Y}|}{|\mathcal{X}| - 1} \leq \frac{|\mathcal{G}|}{|\mathcal{Y}|}.$$

Even if the attacker knows which permutation has been applied, $r - \frac{2^{r-H_2(\mathbf{X})}}{\ln(2)}$ bits of entropy are still obtained.

Thus, if the pool initially contains k bits of *Rényi* entropy and $m \leq k$ bits are extracted by means of the output function, Theorem 1 means that the entropy of the output is greater than $m - \frac{2^{m-k}}{\ln(2)}$, i.e. *is close to* m, which is the desired property.

4 Security Discussion

4.1 Security Requirements

PRNG with entropy inputs must meet several security requirements:

Sound entropy estimation: The PRNG must be able to correctly estimate if enough entropy has been collected to guarantee that an attacker who was not able to observe the input cannot guess the output efficiently.

Pseudorandomness: It must be impossible to compute the content of the internal state and/or to predict future outputs from current outputs of the generator. Moreover, an attacker with a partial knowledge/control of the entropy sources, should be unable to recover the internal state and/or corresponding future outputs.

The next two requirements express resilience against cryptanalytic attacks, and make the assumption that the attacker has had knowledge of the internal state at a specific time. It should be noted in this regard that the Linux PRNG is run entirely in the kernel, which makes it more difficult to access its internal state, in comparison with other software PRNGs such as the Windows PRNG [DGP09].

Forward security: An attacker with knowledge of the current internal state should be unable to recover the previous outputs of the generator (backtracking resistance). Forward security means that knowledge of the internal state provides no information on previous states, even if the state was not refreshed by new entropy inputs. Backtracking resistance can be provided by ensuring that the output function is one-way [BK07]. The design of such generators generally relies on a one-way output function with feedback.

Backward security: Assuming enough future entropy inputs, an attacker should be unable to predict the future outputs of the generator (prediction resistance) based on the knowledge of its current internal state. The output function is deterministic, therefore if an adversary knows the internal state, he will be able to predict the corresponding output as well as future outputs until enough entropy is used to refresh the pool. Consequently, backward security can only be provided if the internal state is effectively reseeded between the requests [BK07].

In the case of the Linux PRNG, the internal state is made of three pools. Forward and backward security must be provided if an attacker has knowledge of one or several pools. Moreover, an attacker must be unable to recover the content of the input pool from the output pools. This requirement is made necessary by the fact that `/dev/urandom` allows the generation of an arbitrarily large amount of bits without new inputs from the input pool, which makes it theoretically easier to guess the content of the nonblocking pool.

4.2 Sound entropy estimation

4.2.1 Empirical validation

As empirical data to test the estimator, we gathered more than 7M samples from the user input source (representing more than 200.000 single samples keyboard and mouse events), by modifying the `add_input_randomness()` function in the Linux PRNG to log the values associated to each input event through the kernel's `printk` interface. The corresponding log messages where then collected in user space by a `syslog` daemon which wrote them to a dedicated file. We believe these measurements to have had little to no impact on the input samples, since they were done outside of any critical section within the PRNG and thus did not delay the handling of new events significantly. This modified kernel was left running on a desktop system through several weeks of daily use to generate the empirical data.

Let N be the number of samples, we consider the specific outcome $\delta_1^{[1]}, \delta_2^{[1]}, \ldots, \delta_{N-1}^{[1]}$. Let $\hat{p}_\eta = \#\{i : \delta_i^{[1]} = \eta\}/(N-1)$ be the empirical frequency of η in the given sequence. We then computed the following values over the empirical data (for precise definitions of entropies see Appendix A):

- $\frac{1}{N-3}\sum_{i=2}^{N-1}\hat{H}_i^{[3]}$: the average estimated entropy as computed by the Linux PRNG,
- $\hat{H} = -\sum_{\eta=0}^{D-1}\hat{p}(\eta)\log_2(\hat{p}(\eta))$: the Shannon entropy based on the empirical frequencies,
- $\hat{H}_{\min} = -\log_2\left(\max_{0\le\eta\le D-1}(\hat{p}(\eta))\right)$: the Min-entropy based on the empirical frequencies, and
- $\hat{H}_2 = -\log_2\sum_{\eta=0}^{D-1}\hat{p}(\eta)^2$: the Rényi entropy based on the empirical frequencies.

The results, when considering the differences not only in the jiffies counts but also in the clock cycles and the `num` values of events, are given in Table 1.

	jiffies	**cycles**	**num**
$\boldsymbol{\frac{1}{N-3}\sum_{i=3}^{N-1}\hat{H}_i^{[3]}}$	**1.85**	10.62	5.55
$\hat{H}$	3.42	14.89	7.31
$\hat{H}_{\min}$	0.68	9.69	4.97
$\hat{H}_2$	1.34	11.29	6.65

Table 1: Comparison of different estimators.

We remark that the average value of $\hat{H}_i^{[3]}$ is always smaller than the empirical Shannon entropy and that the entropy of the jiffies is lower than the entropy of the clock cycles and than that of the `num` value. The Linux entropy estimator is therefore pessimistic, as intended.

4.2.2 Different Levels of Delta

The estimator used in the Linux PRNG is based on three levels of differences. In the following we consider an alternate estimator using k levels of differences. For this we define for $i \geq k-1$:

$$\hat{H}_i^{[k]} = LOG_2\Big[\min\Big(|\delta_i^{[1]}|, \ldots, |\delta_i^{[k]}|\Big)\Big] \tag{5}$$

where $\delta_i^{[j]} = \delta_i^{[j-1]} - \delta_{i-1}^{[j-1]}$ for $1 \leq j \leq k$. In Figure 6, we examine its expectation for different values of k under the theoretical assumption that the $\boldsymbol{\Delta}_i^{[1]}$'s are uniform, independent and identically distributed. This expectation does not change much for $2 \leq k \leq 5$, which might lead us to deduce that $k = 2$ could be sufficient.

However, we note that the empirical distribution of our sample data is far from uniform, and most of the differences are smaller than 150, see Figure 7.

The average values of the estimator for different values of k, computed on the empirical data, are listed in Table 2. We remark that there is a big difference between $k = 3$ and $k = 4$. This suggests some correlations in the data (which is to be expected in the case of user inputs) and supports the idea of considering several levels of deltas.

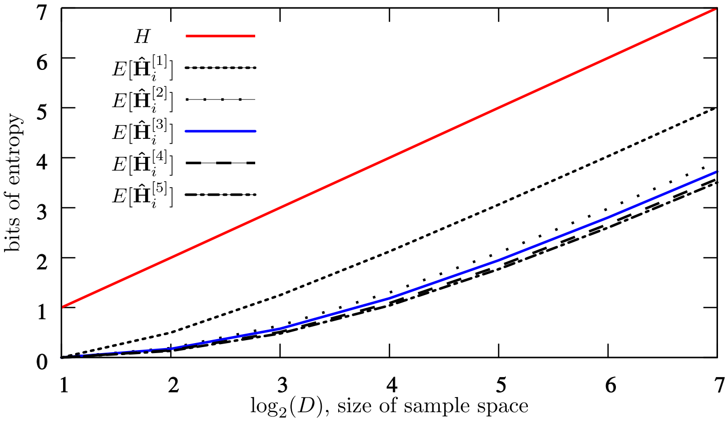

Figure 6: $E[\hat{\mathbf{H}}_i^{[k]}]$ for $\mathbf{\Delta}_i$ uniformly distributed.

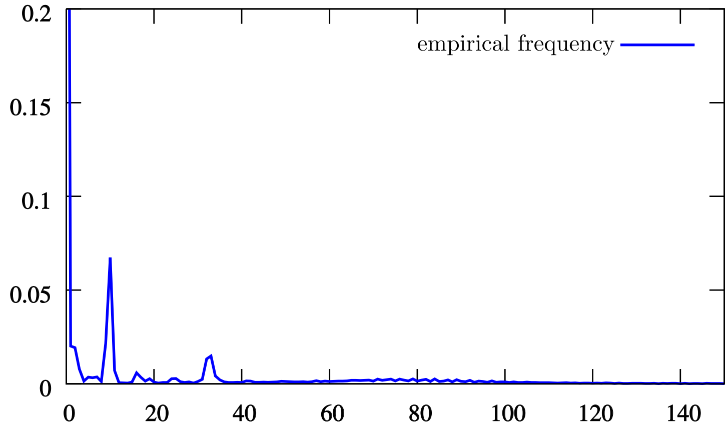

Figure 7: Extract of the empirical distribution of jiffies differences.

4.2.3 Alternatives

We shortly discuss some alternative entropy estimators and the reasons why they cannot be used in the case of the Linux PRNG, based on the constraints presented in Section 2.3.

In the case of a physical source with a known, fixed probability distribution it is sufficient to check if it works correctly as in [Sch01]. However due to Condition 1 from Section 2.3, this method cannot be used. Many estimators use the empirical frequencies or can be computed only on the whole data set [BDGM97, Pan03] which is not consistent with Condition 5. An estimator is proposed in [BL05], which uses the transition frequency of bits. However, it only works for binary sources and thus contradicts Condition 3. Some methods use compression rates to estimate the entropy. However these methods either need a tree of a size depending on the sample space [Vit87] and thus violate Condition 4, or require that a variable number of events be grouped before giving an estimate [WZ89, Gra89, KASW98], which does not fulfill Condition 5.

The main advantages of the estimator implemented in the Linux PRNG are that it is fast, takes little memory, works on non-binary data and gives an information estimate for each event. None of the considered alternatives fit these requirements. The main disadvantage is that there is no theoretical connection to any entropy definition. While not perfect, the estimator gives some notion of the variations of the input data and therefore of the changing behavior of the

	$\hat{H}$	$\frac{1}{N-k}\sum_{i=k}^{N-1}\hat{H}_i^{[k]}$			
		$k=1$	$k=2$	$k=3$	$k=4$
jiffies	3.42	1.99	1.99	1.85	1.47
		$k=5$	$k=6$	$k=7$	$k=8$
jiffies		1.36	1.27	1.10	0.99

Table 2: Different levels of delta (empirical data).

system. Thus, in the absence of suitable alternatives, it can be seen as a good compromise.

4.3 Pseudorandomness

Analysis without entropy input. In this case, we consider the generator without entropy input, and thus as a deterministic PRNG. It uses the output function to generate outputs and the mixing function for feedback. There are general models to construct provably secure PRNG using a one-way function [GKL93, HILL99]. Several deterministic PRNG using Sha-1 have been proposed to date, see for example the DSA generator [FIP00, Appx. 3], (analyzed by Schneier et al. [KSWH98]) or the Hash_DRBG with Sha-1 in [BK07]. The hash function is traditionally considered to be a pseudorandom function in classical PRNG design.

In contrast to those examples, the Linux PRNG updates only part of its internal state (640 bits for each generated 80 bits) and applies a complex feedback function by means of the mixing function. Thus, it cannot be described in one of the existing models. Nevertheless, to our knowledge there is no realizable attack on this PRNG without knowledge of the internal state. More precisely, considering that the Sha-1 hash function is one-way (which is not disproven by recent attacks on this function Sha-1 [WYY05]), the adversary cannot recover the content of the corresponding output pool if he knows only the outputs of the PRNG. In addition, the folding operation helps in avoiding recognizable patterns: the output of the hash function is not directly recognizable from the output data. For an optimal hash function, this step would not be necessary and could be replaced by a simple truncation. The same reasoning also applies to the content of the input pool if the attacker has access to the data transfered from it into either one of the output pools, since those transfers rely on the same output function.

Consequently, the Linux PRNG without entropy input is assumed to be secure if the internal state is not compromised and if the three pools are initialized by an unknown value.

Input based attacks. From the definition of the mixing function and Lemma 1, we know that if an attacker controls the entropy input but has no knowledge of the input pool, he cannot reduce the entropy of that pool. Consequently the behavior of the PRNG corresponds to a deterministic PRNG, and the security analysis above still applies.

4.4 Forward Security

In the context of the Linux PRNG, if we assume that an attacker has knowledge of both the output pool and the input pool, then he knows the previous state except for the 160 bits which were fed back during the last output generation. Without additional information, the only generic attack has an overhead of 2^{160} and produces 2^{80} solutions.

4.5 Backward Security

Protection against backward attacks, as described in Section 4.1, relies on transferring and collecting procedures. *Transferring* k bits of entropy from state S_1 to state S_2 means that after generating data from the unknown state S_1 and mixing it into the known state S_2, guessing the new state value S_2 would cost on average 2^{k-1} trials for an attacker. *Collecting* k bits of entropy means that after processing the unknown data into a known state S_1, guessing the new state value S_1 would cost on average 2^{k-1} trials for an observer.

We consider the case where the attacker was able to learn the internal state at a given time and tries to keep this knowledge by guessing the new input to the pool and checking the guess by observing the output, with two possible attack scenarios.

In the first one, the attacker knows the output pool, but not the input pool. In this case, after transferring $k \geq 64$ bits of entropy data from the input pool, the attacker looses the knowledge of k bits in the state. No output is generated before all of the k bits are mixed in, therefore the only generic attack has an overhead of 2^{k-1}. If the input pool does not contain enough entropy (less than 64 bits for `/dev/random` and less than 64+128 bits for `/dev/urandom` by default, see Footnote 3) no bits are transferred. In that case, the attacker keeps his knowledge of the output pool until enough entropy is collected into the input pool to transfer at least 64 bits. This process prevents 64 bits with low entropy from being transferred from the input pool which would facilitate a guess-and-determine attack. Thus the generator has a resistance of 64 bits by default against this kind of attacks.

In the second scenario, the attacker has knowledge of both the output pool and the input pool. If k bits of entropy are collected before the adversary sees the output, the complexity of guessing the input is 2^{k-1} on average. As long as the entropy counter in the input pool is high enough this can happen for $k < 64$. However, this will reduce the entropy counter of the input pool, which will eventually be low enough that at least $k \geq 64$ bits of entropy must be collected between two transfers to the output pool. This again leads to a default resistance of 64 bits.

4.6 Conserving the Entropy

Barak and Halevi presented a theoretical model of a PRNG with entropy inputs in [BH05]. Their work suggests the use of a deterministic extractor[5] in the entropy extraction procedure, immediately after the entropy source. The authors consider that the collected data has full entropy. The security of this model assumes regular inputs with minimal entropy. This is a problem if the input samples are corrupted or partially manipulated by an attacker.

As discussed in Section 3.3.1, when extracting $m \leq k$ bits from a pool containing k bits of *Rényi* entropy, we can assume the entropy of the output to be greater than $m - \frac{2^{m-k}}{\ln(2)}$. One problem is that the Rényi entropy is always smaller or equal to Shannon's entropy, with equality only for the uniform distribution. However, at least in our empirical data, the Rényi entropy of the combination of the three values per event is always much higher than the estimated Shannon entropy of the jiffies.

5 Changes to Previous Versions

In this section we outline the most significant changes to the Linux PRNG since the version analyzed in [GPR06].

Earlier versions provided only two 32-bit words as entropy input for each event, namely the `num` value specific to the event and either the jiffies count *or* the cycle count, depending on the source. This was changed to the current three words as far back as version 2.6.12 (see Section 2.2).

[5] Extractors in the mathematical sense which were created for derandomization procedure. For a survey, see [Sha02].

To improve the resistance against input based attacks the current version mixes the data from the entropy sources only into the input pool. In previous versions, the samples were also mixed directly into the blocking pool when the entropy counter of the input pool was at its maximum. Moreover, in the previous version, samples were mixed by blocks of 32 bits. This lead to inconsistencies when data were transferred from the input pool to the output pool in blocks of 80 bits, which is not a multiple of 32. This inconsistency was corrected by switching to single byte operations in Linux 2.6.26 (see Section 2.3)

In the current version, `/dev/urandom` always leaves, in its default configuration, at least 16 bytes of entropy in the input pool. Moreover, no transfer is done when the input pool cannot deliver more than 8 bytes of entropy. These two changes were introduced in early 2005 to avoid denial of service attacks on `/dev/random` through repeated reading on `/dev/urandom`, as mentioned in [GPR06], and to increase the backward security. In the previous version the only limit to the number of transferred bytes from the input pool to the output pool was the entropy counter of the input pool. If the input pool contained one byte of entropy, a single byte would have been transferred. Thus the backward security was only 8 bits in this case. (see Sections 2.4 and 4.5)

In the previous version, one word was fed back into the pool after each application of the compression function to 16 words, while hashing the whole pool. This allowed a forward attack [GPR06]. More precisely, Gutterman et al. proposed a method to reverse the pool with an overhead of 2^{96} computations of Sha-1 compression function or 2^{64} computations in some cases, instead of 2^{160}. The attack basically worked as a *divide and conquer* attack, using the fact that the feedback function mixed back only one word in each iteration of the hashing process. The new version, since Linux 2.6.26, hashes the whole pool before it feeds back any data, and is thus resistant to such attacks, as discussed in Sections 4 and 3.3.

6 Conclusion

This paper presents the Linux PRNG for kernel versions 2.6.30.7 and up. We detail the main changes since the analysis of [GPR06], and their security impacts. Our analysis shows that the design of the current version of the PRNG allows it to reach a good level of security (pseudorandomness, backward and forward security, entropy conservation). We also provide a detailed analysis of the main components of the PRNG, for which we point out a few weaknesses and suggest some evolutions.

First, while the comments in the code suggest that the mixing function is based on a twisted GFSR, our analysis shows that this is not the case, and that the characteristic polynomial used is not irreducible over $GF(2^{32})$. A few simple changes in the feedback function could provide a full period for the LFSR with zero input.

Secondly, we regret the lack of connection of the entropy estimator with any entropy definition, but our empirical testing suggests that it nevertheless works reasonably well for unknown data statistics. We investigate possible alternatives for this estimation of the entropy, including simpler or more complex variations of the same approach, but find them to present either no benefit or significant disadvantages when compared with the current estimator.

As a final remark, we note that the Linux PRNG is based on the Sha-1 hash function, for which the security status could be debatable. Modifying the PRNG to make use of a newer hash function, for example Sha-3, would require a significant change of the design, and an investigation of the performance implications to the Linux kernel as a whole.

References

[BBR88] Charles H. Bennett, Gilles Brassard, and Jean-Marc Robert. Privacy amplification by public discussion. *SIAM J. on Comp.*, 17:210–229, 1988.

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust. The magma algebra system i: the user language. *J. Symb. Comput.*, 24:235–265, 1997.

[BDGM97] J. Beirlant, E. J. Dudewicz, L. Györfi, and E. C. Meulen. Nonparametric entropy estimation: An overview. *Int. J. Math. Stat. Sci.*, 6:17–39, 1997.

[BH05] Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In *ACM Conf. on Comp. and Communications Sec. - CCS 2005*, pages 203–212, 2005.

[BK07] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators (revised). Technical Report SP800-90, NIST, March 2007.

[BL05] Marco Bucci and Raimondo Luzzi. Design of testable random bit generators. In *CHES 2005*, volume 3659 of *LNCS*, pages 147–156, 2005.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers.* O'Reilly Media, Inc., 3 edition, 2005.

[CT06] Thomas M. Cover and Joy Thomas. *Elements of Information Theory.* Wiley-Interscience, 2 edition, 2006.

[DGP09] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. Cryptanalysis of the random number generator of the Windows operating system. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009.

[ESC05] D. Eastlake, J. Schiller, and S. Crocker. Randomness requirements for security. RFC 4086 (Best Current Practice), June 2005.

[FIP00] Digital signature standard (DSS). Technical Report FIPS PUB 186-2, NIST, January 2000.

[FS03] Niels Ferguson and Bruce Schneier. *Practical Cryptography.* John Wiley & Sons, 2003.

[GKL93] Oded Goldreich, Hugo Krawczyk, and Michael Luby. On the existence of pseudorandom generators. *SIAM J. Comput.*, 22(6):1163–1175, 1993.

[GPR06] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the Linux random number generator. In *Proceedings of the 2006 IEEE Symp. on Sec. and Privacy*, pages 371–385, 2006.

[Gra89] Peter Grassberger. Estimating the information content of symbol sequences and efficient codes. *IEEE Trans. IT*, 35(3):669–675, 1989.

[Gut98] Peter Gutmann. Software generation of practically strong random numbers. In *USENIX Security Symposium - SSYM'98*, pages 243–257, 1998.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. on Computing*, 28:12–24, 1999.

[JK99] Benjamin Jun and Paul Kocher. The Intel random number generator. Technical Report White paper prepared for Intel Corporation, Cryptography Research, Inc., 1999.

[KASW98] Ioannis Kontoyiannis, Paul H. Algoet, Yuri M. Suhov, and A. J. Wyner. Nonparametric entropy estimation for stationary processes and random fields, with applications to English text. *IEEE Trans. IT*, 44(3):1319–1327, 1998.

[KSF99] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator. In *SAC'99*, volume 1758 of *LNCS*, pages 13–33, 1999.

[KSWH98] John Kelsey, Bruce Schneier, David Wagner, and Cris Hall. Cryptanalytic attack on pseudorandom number generators. In *FSE'98*, volume 1372 of *LNCS*, pages 168–188, 1998.

[LN97] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. Cambridge University Press, 2 edition, 1997.

[MK92] Makoto Matsumoto and Yoshiharu Kurita. Twisted GFSR generators. *ACM Trans. on Modeling and Comp. Simulation*, 2(3):179–194, 1992.

[MT09] Matt Mackall and Theodore Ts'o. random.c – A strong random number generator, September 2009. /driver/char/random.c in Linux Kernel 2.6.30.7, `http://www.kernel.org/`.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Inc., 1 edition, 1996.

[Pan03] Liam Paninski. Estimation of entropy and mutual information. *Neural Computation*, 15(6):1191–1253, 2003.

[Sch01] Werner Schindler. Efficient online tests for true random number generators. In *CHES 2001*, volume 2162 of *LNCS*, pages 103–117, 2001.

[Sha02] Ronen Shaltiel. Recent developments in explicit constructions of extractors. *Bulletin of the ACS*, 77:67–95, 2002.

[SS03] André Seznec and Nicolas Sendrier. HAVEGE: A user-level software heuristic for generating empirically strong random numbers. *ACM Trans. Model. Comput. Simul.*, 13(4):334–346, 2003.

[Vit87] Jeffrey Scott Vitter. Design and analysis of dynamic Huffman codes. *J. ACM*, 34(4):825–845, 1987.

[WC81] Mark N. Wegman and J. Lawrence Carter. New hash functions and their use in authentification and set equality. *J. of Comp. and Syst. Sciences*, 22:265–279, 1981.

[WYY05] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *CRYPTO 2005*, volume 3621 of *LNCS*, pages 17–36, 2005.

[WZ89] Aaron D. Wyner and Jacob Ziv. Some asymptotic properties of the entropy of a stationary ergodic data source with applications to data compression. *IEEE Trans. IT*, 35(6):1250–1258, 1989.

A Entropy

We use three different entropy definitions: the *Shannon entropy*, which is a measure of the number of binary questions needed on average to guess the value of a random variable, the *Rényi entropy* of order 2, which is a measure of the probability of having a collision in two elements, and the Min-entropy, which is a lower bound of all the other entropy measures. The use of Shannon entropy is implied when no other precision is given.

Definition 3. *[CT06] For a random variable* $\mathbf{X}$ *with probability distribution* $p^{\mathbf{X}}$ *the* Shannon entropy $H(\mathbf{X})$ *the* Rényi entropy of order 2 $H_2(\mathbf{X})$ *and the* Min-entropy $H_{min}(\mathbf{X})$ *are defined as*

$$
\begin{aligned}
H(\mathbf{X}) &= -\sum_{\eta\in\mathcal{X}} p^{\mathbf{X}}(\eta)\log_2 p^{\mathbf{X}}(\eta)\\
H_2(\mathbf{X}) &= -\log_2 \sum_{\eta\in\mathcal{X}} (p^{\mathbf{X}}(\eta))^2\\
H_{min}(\mathbf{X}) &= -\log_2 \max_{\eta\in\mathcal{X}} p^{\mathbf{X}}(\eta)
\end{aligned}
$$

It holds that $H_{min}(\mathbf{X}) \leq H_2(\mathbf{X}) \leq H(\mathbf{X})$, *with equality if and only if* $\mathbf{X}$ *is uniformly distributed.*

The conditional variants of the Shannon and Rényi entropy are given, by

$$
\begin{aligned}
H(\mathbf{X}|\mathbf{Y}) &= \textstyle\sum_{\kappa\in\mathcal{Y}} p^{\mathbf{Y}}(\kappa)H(\mathbf{X}|\mathbf{Y}=\kappa) \text{ and}\\
H_2(\mathbf{X}|\mathbf{Y}) &= \textstyle\sum_{\kappa\in\mathcal{Y}} p^{\mathbf{Y}}(\kappa)H_2(\mathbf{X}|\mathbf{Y}=\kappa).
\end{aligned}
$$

B Pseudocode

For the sake of clarity, we give the pseudocode for the mixing function, the output function, the processing of an input event and the generation of random bytes. To provide a single description valid for both the input and output pools we use n to denote the size of the pool in 32-bit words and $(0, i_1, \ldots, i_4, n-1)$ to describe the feedback positions of the mixing function. For the **input pool**, we have $n = 128$ and $(0, i_1, \ldots, i_4, n-1) = (0, 24, 50, 75, 102, 127)$ as for the **output pool**, we have $n = 32$ and $(0, i_1, \ldots, i_4, n-1) = (0, 6, 13, 19, 25, 31)$. We use $b \leftarrow \text{byte}[n]$, to denote that b is a byte array of size n. By $b[i]$ and $b[i \ldots j]$ we mean, respectively, the ith element of b and the elements $b[i], \ldots, b[j]$. All indices are taken modulo the array size.

B.1 The Mixing Function: mix(*pool*, *input*)

For any byte y, let word32(y) denote the extension of y to a 32-bit word. For any word w, $w <<< rot$ is the bitwise rotation of w to the left by rot bits.

Require: The n pool words: $pool \leftarrow \text{word}[n]$
Require: The m input bytes: $input \leftarrow \text{byte}[m]$
Require: Last stored rotation factor: $rot \in \{0, \ldots, 31\}$
Require: Last input position: $i \in \{0, \ldots, n-1\}$

for $j = 0$ to $m-1$ **do**
  $i \leftarrow i - 1 \pmod{n}$
  $w \leftarrow \text{word32}(input[j])$
  $w \leftarrow w <<< rot$

$$\begin{aligned} w \leftarrow \quad w \quad & \oplus \; pool[i+1] && \oplus \; pool[i+i_1+1] \\ & \oplus \; pool[i+i_2+1] && \oplus \; pool[i+i_3+1] \\ & \oplus \; pool[i+i_4+1] && \oplus \; pool[i] \end{aligned}$$

  $pool[i] \leftarrow w$
  if $i = 0$ **then**
    $rot \leftarrow rot + 14 \pmod{32}$
  else
    $rot \leftarrow rot + 7 \pmod{32}$

B.2 The Output Function: $\text{out}(pool, k)$

The output function produces k bytes from the pool. The Sha-1 compression function is denoted by $\text{sha1}(cv, m)$. It takes a 20-byte chaining value cv and a 128-byte message m and outputs a new 20-byte chaining value. The 20-byte initial value of the Sha-1 hash function is denoted by IV_{sha1}. The fold function folds 20 bytes to 10 bytes as explained in Section 3.3 and $\text{trunc}(\cdot, r)$ is the truncation to r bytes.

Require: The n pool words: $pool \leftarrow \text{word}[n]$
Require: Number of requested bytes: k
Require: Last input position: $i \in \{0, \ldots, n-1\}$
Require: The output buffer: $res \leftarrow \text{byte}[k]$
  $b \leftarrow \text{byte}[20]$
  $j \leftarrow 0$
  while $j < k$ **do**
    $b \leftarrow IV_{\text{sha1}}$
    for $\ell = 0$ to $n/16 - 1$ **do**
      $b \leftarrow \text{sha1}\left(b, pool[16\ell \ldots 16\ell + 15]\right)$
    $\text{mix}(pool, b)$ {Changes the pool and i}
    $w \leftarrow \text{word}[16]$
    for $\ell = 0$ to 15 **do**
      $w[\ell] = pool[i - \ell \pmod{n}]$
    $b \leftarrow \text{sha1}(b, w)$
    $r \leftarrow \min(10, k - j)$
    $res[j \ldots j + r - 1] \leftarrow \text{trunc}(\text{fold}(b), r)$
    $j \leftarrow j + 10$

B.3 Adding an Event to the Pool: $\text{add}(pool, event)$

This algorithm is used to add the data of an event to the input pool and to update the entropy counter. Each event $event = (source, jif, cyc, num)$ contains the information of its source, and three values containing the current jiffies count, cycle count and an additional number. We denote by $\text{entr}(source, jif)$ the estimation of the entropy based on the current jiffies count and the previous values stored for this specific source.

Require: The 128 input pool words: $pool \leftarrow \text{word}[128]$

Require: Jiffies count of the event: jif
Require: Cycles count of the event: cyc
Require: Number specific to the event: num
Require: Source of the event: $source$
Require: Entropy counter of the input pool: $h \in \{0, \dots, 4096\}$
  $\text{mix}(pool, jif)$
  $\text{mix}(pool, cycles)$
  $\text{mix}(pool, num)$
  $h \leftarrow h + \text{entr}(source, jif)$
  $h \leftarrow \max(h, 4096)$

B.4 Generating Random Bytes: $\text{gen}(pool, k)$

The following pseudo code describes the complete procedure for generating k bytes from one of the output pools, including entropy counter check.

Require: The 32 pool words: $pool \leftarrow \text{word}[32]$
Require: Number of requested bytes: k
Require: Output buffer: $res \leftarrow \text{byte}[k]$
Require: Entropy counter of the pool: $h \in \{0, \dots, 1024\}$
Require: Input pool: $inpool \leftarrow \text{word}[128]$
Require: Entropy counter of the input pool: $h_{\text{I}} \in \{0, \dots, 4096\}$
  if $h < 8k$ **then**
    $\ell \leftarrow \min(\max(k, 8), 128)$ {Guarantees $8 \leq \ell \leq 128$}
    if `/dev/urandom` **then** {Leave 16 bytes of entropy}
      $\ell \leftarrow \min(\ell, \lfloor h_{\text{I}}/8 \rfloor - 16)$
    else {`/dev/random` }
      $\ell \leftarrow \min(\ell, \lfloor h_{\text{I}}/8 \rfloor)$
    if $\ell \geq 8$ **then** {Enough entropy for transfer}
      $trans \leftarrow \text{byte}[\ell]$
      $trans \leftarrow \text{out}(inpool, \ell)$
      $\text{mix}(pool, trans)$
      $h \leftarrow h + 8\ell$
      $h_{\text{I}} \leftarrow h_{\text{I}} - 8\ell$
  if `/dev/random` and $h < 8k$ **then** {Limited output}
    $k' \leftarrow \lfloor h/8 \rfloor$
    $res[0 \dots k' - 1] \leftarrow \text{out}(pool, k')$
    $h \leftarrow h - 8k'$
    while input pool has not enough entropy to reseed **do**
      Wait
    $res[k' \dots k] \leftarrow \text{gen}(pool, k - k')$
  else
    $res \leftarrow \text{out}(pool, k);$
    $h \leftarrow \max(0, h - 8k)$