A Survey on the Linux Random Number Generator

Shuai LI*
* Department of Computer Science
University of Calgary, Alberta, Canada
shuai.li1@ucalgary.ca

Abstract— **The Linux random number generator (LRNG) produces random data for many security related applications and protocols. The generator is part of the Linux kernel open source project which has different versions since the original one. The survey describes the generator of Linux kernel version 2.6.30.7. The study provides a comprehensive analysis of all functions in LRNG as well as a security analysis against cryptographic attacks.**

I. Introduction

Random number generation has become a basis of modern computational where security is involved. The random data serves as keys in crypto systems, such as the secret key in DES encryption system, the prime p,q in RSA encryption and digital signature. Cryptographic protocols require random data. Examples include big integers in Diffie-Hellman key exchange protocol and nonces in challenge response authentication protocol. The security of these systems, protocols depends on the unpredictable random bits. With the low quality random bits, the attacker might guess these bits and then break the cryptographic system.

There are two types of random number generator. True random number generator(TRNG), pseudorandom number generator (PRNG). TRNG exploits the randomness from some entropy source such as natural phenomena[3] and human game play[1], however, such entropy input may produce bits that are biased (the probability of 1 is not 0.5) or correlated (the output of next bit depends on the previous bit). Passing a de-skewing (processing function) is necessary to remove biases and correlations. Although TRNG outputs non-periodic and non-deterministic bits, it often has to wait until enough entropy is collected. A pseudorandom number generator takes a seed value which is a random bits and uses a deterministic algorithm to produce sequences that appear to be random. PRNGs are more efficient when generating a large amount of sequences. PRNGssecure against cryptographic attacks are cryptographically secure pseudorandom number generators (CSPRNG). CSPRNG requires that i) the generators output bits are unpredictable to an attacker without knowledge of internal state of the generator. ii) An attacker which learn the state of the generator at a certain time cannot compromise the outputs at previous times and all future outputs. The Linux random number generator is a CSPRNG[5][2].

LRNG exploits randomness from system events inside the kernel as entropy sources. The entropy is accumulated from these entropy sources into the internal state of the generator. A post-processing procedure outputS random bits and update the internal state. The generator is used by internal kernel functionalities and by user calls to its APIs. Linux provides two character device interfaces to read random data. /dev/random and /dev/urandom. The difference between these two APIs are the level of randomness of the random bits and the resulting delay. /dev/random is intended for applications which requires a small number of extreme secure bits. Reading from this device limits the number of generated bits depending on entropy in LRNG. The second device /dev/urandom generates less secure bits but never limit the random data generation.

The remainder of the report is organized as follows. In section II, we provide background knowledge of randomness measurement and entropy. Section III gives a high-level structure of the LRNG. Section IV includes discussions of each building blocks and a complete procedure of generating random data from each output pool. A security analysis of version 2.6.30.7 is included in section V.

II. Background

A. Measure of randomness

Once random bits are generated. It is important to determine how random the sequence is. One of the widely deployed measurement is NIST statistical test[6]. The NIST statistical test suite contains 15 tests to measure the randomness of the generated binary sequence. Each statistical test determines whether the sequence has a certain statistic property that expected for a random sequence. The conclusion of each test is not definite, but rather probabilistic. If a sequence passes all tests, there is no gurantee that it is indeed produced by a random bit generator.

Hypothesis Testing. A null hypothesis, H_0 is an assertion that a tested sequence is random. In contrast, the alternative hypothesis, H_a is an assertion that the tested sequence is not random. Each test has a randomness statistic to determine the acceptance or rejection of the null hypothesis. Given a test sequence and a test, a theoretical reference distribution of this statistic under an assumption of randomness is determined by mathematical methods. From this reference distribution, a critical value is determined (specific to the type of test, and the sensitivity of the test). During a test, a test statistic value is computed on the sequence being tested.. This test statistic value is compared to the critical value. If the test statistic value exceeds the critical value, the null hypothesis for randomness is rejected. Otherwise, the null hypothesis is accepted.

B. Entropy

Entropy measures the uncertainty (in bits) of predicting the value of a random variable.

Shannon Entropy: Shannon entropy, denoted as $H(X)$ is defined on a discrete random variable X [4]. It is the average (expected) information from all outcomes. $p(x)$ is the probability of x.

$$H(X) = -\sum_{x \in X} p(x) log_2(p(x))$$

Min Entropy: The min entropy is a lower bound of Shannon Entropy. It is denoted as $H_\infty(X)$. If the probability distribution of X is $p_1, p_2, p_3, ..., p_m$, min entropy is defined as[4],

$$H_\infty(X) = -log_2(max(p_1, p_2, p_3, ..., p_m)$$

The min entropy captures the refined uncertainty of predicting the value of X when the attacker guesses the value of X with highest probability.

III. General structure

The general structure of LRNG mainly contians three asynchronous procedures as depicted in Figure 1. In the first procedure, the entropy is collected from system events. In the second procedure, the entropy is accumulated into the internal state of LRNG. The last procedure, the output is generated by applying the output function on the internal state and the internal state is updated.

Pools. The internal state of LRNG consists of three pools, input pool, blocking pool and unblocking pool. The entropy input collected from the system events is mixed into input pool first. The size of the input pool is 128 words (4096 bits). The blocking pool and the unblocking pool are output pools. The sizes of both output pools are 32 words. When /dev/random is called, the output data is generated from the blocking pool. When /dev/urandom is called by the user space or get_random_bytes is called by internal kernel, output bits are generated from the unblock pool.

Entropy Inputs an Accumulation. Entropy inputs provides backbones of the security of LRNG. Entropy are collected and mixed into the input pool asychronously and independtly from output generation. TLRNG uses four different sources: mouse and keyboard input, disk I/O operations, and specific interrupts. For each event fed in to LRNG, three 32 bits input value are recorded: a 32 bits value num representing the value and type, a 32-bit value representing the jiffies count which corresponds to the internal kernel counter of timer interrupts since the last kernel boot. a 32 bit value representing the current CPU cycle count. The latter two keeps the time information when the event is mixed into the pool. The difference of jiffies associated with successive events of the same type are used to estimate the entropy provided by the event. More details of the entropy estimation will be discussed in section IV-C.

The num, $jiffies$, $cycle$ for the event are added to the input pool one by one using the mixing functions described in section IV-C. The estimated entropy of the event is added to the input pools entropy counter.

Entropy Counter check. Each pool has an entropy counter which estimate the current entropy in the pool. This is an integer between 0 and the size of the pool in bits. The range of the counter for input pool is [0,4096] while the range of the counter for both output pools are [0, 1024]. Three entropy counters are vital to the correct functioning of LRNG. For entropy accumulation, the entropy counter of input pools is increased by the amount of estimated entropy from the entropy source. For output generation, two different cases are considered separately. One is generating data from /dev/random, the other one is generating data from /dev/urandom. In the first case, /dev/random is called to generate k bytes random data, the blocking pool will request to collect 8k bits of entropy from the input pool. If input pool contains enough entropy, the input pool will extract 8k bits of entropy using the output function and the counter of the input pool decrements by 8k bits. The output of the input pool is mixed into the blocking pool using the same mixing function mentioned above and the counter of blocking pool increasesby 8k bits. Otherwise, the block pool blocks and waits until enough entropy is collected by the input pool. After generating the k bytes random data using the same output function, the counter of the blocking pool decrements by 8k bits. In the second case, /dev/urandom is called to generate k bytes random data. Even if input pool does not contain enough entropy, the nonblocking pool will generate k bytes data. The counter of the non blocking pool is decremented by 8k bits. If the current value of the nonblocking pools counter is 0, the counter value will not change. It is worth to note that the entropy counter of the output pools will generallyremain close to zero since the amount of transferred bit are as many as the amount of output bits.

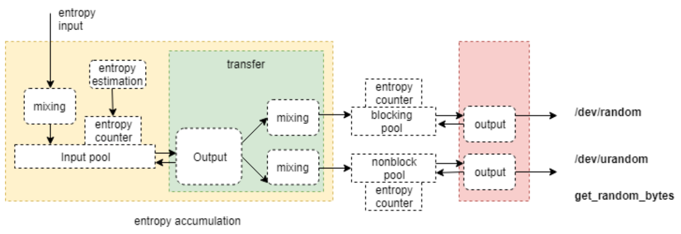

Fig. 1: General Structure of LRNG

Generating output The generator uses the same output function to extract data from the input pool and two output pools. The process of the output function includes three steps: i) updating the pools content, ii) generating random bits, iii) decrementing the entropy counter. The output function is precisely described in section IV-D.

IV. Building Blocks

In this section, we illustrate all building blocks of LRNG. At the end, we give a complete algorithm for generating k bytes random data from each output pool.

A. Entropy Collection

The LRNG collects entropy from different entropy sources, namely mouse and keyboard activity, disk I/0 event and interrupt event. For each entropy event fed to the input pool, three 32 bits valus are recorded: num,which is the type-value specific to the event, $jiffies$ and $cycle$, which records different timing information when the entropy event is mixed into input pool.

num. The num value are different for each event type.

- keyboard event. The num contains the keyboard press, the valid range is [0, 255]. Only 8 out of 32 bits are effective.
- mouse event.

  $$num := (type \ll 4) \oplus code \oplus (code \gg 4) \oplus value$$

  Type describes the event type: pressing or releasing the mouse buttons and starting move or ending move the mouse; code is the mouse button pressed (left, right or middle) or wheel scrolling and the axis of the mouse movement (horizontal or vertical); value is true when mouse buttons are pressed, otherwise false. Value is 1 or 1 for denoting scrolling direction (1 for up, 1 for down) in case of wheel scrolling. Value is the size of movement in case of mouse movement, 10 bits are used for movement. The movement size is main entropy factor for mouse event.In fact only 12 out of the 32 bits are effective.
- disk event. The num consist of major and minor numbers which together define a device in OS.

  $$num := 0x100 + ((major \ll 20)|minor)$$

  Assuming an average machine has no more than 8 disks, only 3 out of 32 bits are effective.
- interrupt event. The num contains the interrupt request channel number, with a valid range of [0,15]. Only 4 out of 32 bits are effective.

jiffies. It represents the jiffy count value of the system at whcih the events mixed into the input pool. Jiffy is the time between two ticks of the system timer interrupt and the jiffy count value represents the total number of ticks of system timer interrupt from the boot time[4].

cycles. It represennts the CPU cycle count value. Cycles are aslo measured from the boot time of the device[4].

B. The Mixing Function

The mixing function is applied when the entropy sample is added into input pool, and when data from the input pool is transferred from the input pool to one of the output pools. Algorithm 1 describes the linear mixing function for a pool of size 32 word. For any byte y, let $extension32(y)$ denote the extension of y to a 32 bits word. For any word w, $w <<< rot$ is the bitwise rotation of w to the left by rot bits.

Entropy is added to pool of size 32 words (blocking pool and nonblocking pool) by running Algorithm 1, and updating the index i of the pool. One byte is mixed at a time. First, the byte w is extended to 32 bits (padding with 0s). Then, it is rotated by a changing factor and applied multiplication in $GF(2^{32})$. w is xored with pool entries $i, i+1, i+7, i+14, i+20, i+26$. Next, the last 3 bits of w choose a table entry which is xored to $(w \gg 3)$. the calculated value is assigned to w. Finally, the pool is updated by xoring w with $pool[i]$. The index i is updated after each iteration. The value of i determines the value of the rotation factor in the nexT round. Every bytes is mixed into the pool with the same manner.

The same mixing function is applied to the input pool with the size of 128 words. However, the polynomial is changed to $x^{128} + x^{103} + x^{76} + x^{51} + x^{25} + x + 1$ and modulo of 128 is calculated instead of 32 . We use this polynomial to update the w (namely, xoring w with pool entries $i, i+1, i+25, i+51, i+76, i+103$).

In[5], it verifies that the mixing function guarantees that if an attacker has no knowledge of the three pools but has complete control of the entropy input, he will gain no addition knowledge of the new state of three pools after execution of the mixing function. Such property provides backbones of the pseudorandomness of the generated data.

Algorithm 1 $mix(pool, input)$

```
Input: the pool[32], m input bytes, last stored rotation
    factor rot, last input position i, twist_table
Output: The input is mixed into the pool one byte at a
    time
1:  for $j = 0\ to\ m-1$ do
2:      $i \leftarrow i - 1(mod\ 32)$
3:      $w \leftarrow extension32(input(j))$
4:      $w \leftarrow w <<< rot$
5:      $w \leftarrow w \oplus pool[i]$
6:      $w \leftarrow w \oplus pool[(i+1)\ mod\ 32]$
7:      $w \leftarrow w \oplus pool[(i+7)\ mod\ 32]$
8:      $w \leftarrow w \oplus pool[(i+14)\ mod\ 32]$
9:      $w \leftarrow w \oplus pool[(i+20)\ mod\ 32]$
10:     $w \leftarrow w \oplus pool[(i+26)\ mod\ 32]$
11:     $w \leftarrow (w \gg 3) \oplus twist\_table[w\ \&\ 7]$
12:     $pool[i] \leftarrow w$
13:     if $i = 0$ then
14:         $rot \leftarrow rot + 14(mod\ 32)$
15:     else
16:         $rot \leftarrow rot + 7(mod\ 32)$
```

C. The Entropy Estimator

The difference of jiffies associated with successive events of the same type are used to estimate the entropy provided by the event. The num and $cycle$ values are not used to estimate the entropy. The entropy is estimated in the following way:

Let t_n denote the timing of the event number n. Define

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

The entropy mixed into the pool by the event is defined to be $log_2(min(|\delta_n|, |\delta_n^2|, |\delta_n^3|)_{[19-30]})$ where $S_{[a-b]}$ denotes bits a to b (inclusive) of S. The entropy is also bounded by a maximum output of 11 since $log_2 2^{11} = 11 bits$. Although each entropy input consists of a num, jiffies and cycle values, the LRNG entropy estimation is based only on jiffies which leads to a pessimistic evaluation. In[5], it shows that LRNG entropy estimator is pessimistic by the observation that the average empirical entropy of $jiffies$ is less than that of $cycles$ and num.

D. The output function

The same output function is used to extract transferring bits and generate random bits. The output function is depicted in Figure 2 with the pool size of n.

There are two phases in output generation. Feedback phase and extraction phase. In the feedback phase, a chained SHA-1 is applied to the entire pool, the 5 words output value is mixed into the original pool, resulting an update of the pools state. In the extraction phase, 16 words are selected from the updated pool, and again the 16 words are fed into SHA-1. Finally, the 5 words output value is folded into 10 bytes of data. The output function always generates 10 bytes data each time. If the required amount of bytes is multiple of 10, the output function repeats until all bytes are generated. In the case of the number of requested bytes is not a multiple of 10, the last block is truncated to the length of the missing bytes.

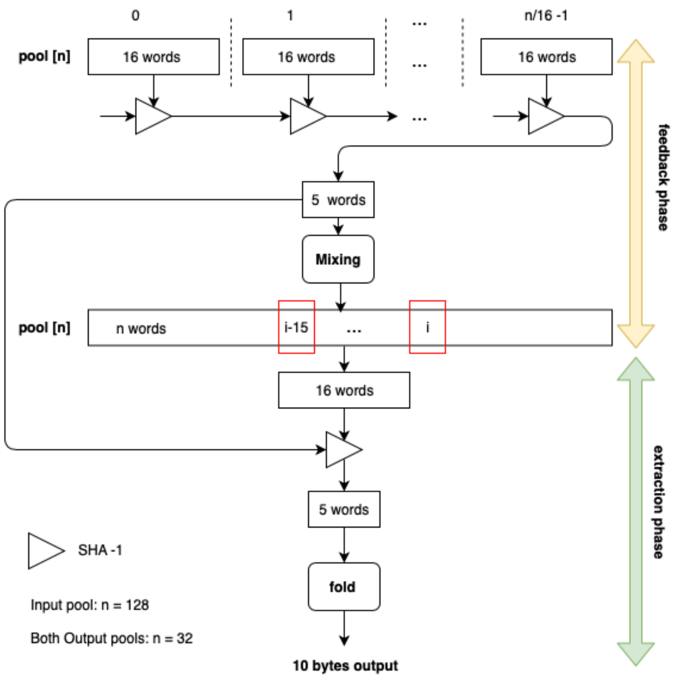

Fig. 2: The output function of LRNG

Feedback Phase. The pool is evenly dividedto blocks, each block has 16 words. The input pool (128 words) has 8 blocks, 2 blocks for both output pools (32 words). The SHA-1 function is denoted as $SHA-1(CV, M)$, It takes 20 bytes of chaining value CV, and message M, output a block with a fixed size of 20 bytes. Beginning at first block of the pool, the content of the block and the SHA-1 initial IV value produce a 20 bytes SHA-1 output. This output, together with the content of block 2 of the pool, are fed into the second SHA-1 function, generating a 20 bytes output as the chaining value for the next SHA-1 function. At the end of the day, a 20 bytes output will be generated from the entire pool. The next step is to mix the 20 bytes into the original pool. The same mixing function in Algorithm 1 is used here. 20 bytes are added into the pool one by one, resulting the 20 times updates of the pool content.

Extraction Phase. As mentioned in section IV-B, execution of mix function results in the index changeas well. In the next step, 16 words before the pool[i] in the updated pool are selected. The SHA-1 is used once again, taking these 16 words and the 20 bytes from the feedback phase and generating 20 bytes. Finally, the output of SHA-1 in the extraction phase is folded to 10 bytes. If $w_{[m...n]}$ denotes the bits m,...,n of the word w, the folding operation of the five words w_0, w_1, w_2, w_3, w_4 isdone by $w_0 \oplus w_3, w_1 \oplus w_4, w_{2[0...15]} \oplus w_{2[16...31]}$ The 10 bytes are accumulated to the random data buffer. Once all random data are generated, the buffer will release the generated random data. The entropy counter of the affected pool decrements by the number of generated bits.

The pseudo code for output function (nbytes are requested from $pool$)is given in Algorithm 2. The $Output()$ function decides how many bytes of data are extracted from the 10 bytes data in the case of nbytes is not a multiple 10.

Algorithm 2 $output(pool, nbytes)$

```
    Input: SHA-1 initial IV, pool size n, nbytes
    Output: nbytes random data or transferring bits
 1: while $nbytes > 0$ do
 2:     $b \leftarrow IV$
 3:     for $l = 0\ to\ \frac{n}{16} - 1$ do
 4:         $b \leftarrow$SHA-1$(b, pool[16l...16l + 15])$
 5:     $mix(pool, b)$      // Change pool and index i.
 6:     $p \leftarrow word[16]$
 7:     for $l = 0\ to\ 15$ do
 8:         $p[l] = pool[i - lmod\ n]$
 9:     $b \leftarrow$ SHA-1$(b, p)$
10:     $Output(folding(b),\ min(nbytes, 10))$
11:     $nbytes = nbytes - min(nbytes, 10)$
```

E. Complete Procedure

When k bits random dataare requested, the generator first checks whether there is enough entropy in output pool according to the entropy counter. If there is enough entropy in output pool, k bytes are generated from this pool and the entropy counter for that pool decreases by 8k bits, Otherwise, output pool requests input pool to transfer the missing entropy. After a successful transfer, the counter of the input pool decreases certain bits while the counter of the output pool increases the same amount of bits. Then the k bytes are generated from the output pool, the entropy counter of the output pools decreases by 8k bits. The entropy transfer happens much more frequently because the entropy counter of the output pool will generallyremains close to zero. The amount of transferred bit are always as many as (blocking pool)or less than (nonblocking pool) the amount of output bits.

We consider the whole procedure for generating k bytes from each of the output pools.

Generating data from blocking pool. k bytes random data are requested to generate by /dev/random. Let h_i be the current entropy counter of the input pool, h_o be the current entropy counter of the blocking pool. If $h_O < 8K$, a transfer request is sent to the input pool to transfer $k - \lfloor h_o/8 \rfloor$ bytes entropy (usually$\lfloor h_o/8 \rfloor = 0$). Then input pool extracts k bytes, $k = min(\lfloor h_1/8 \rfloor,\ k - \lfloor h_o/8 \rfloor)$. This means that the input pool never transfer more bits than its entropy counter allows. Moreover, no transfer is done when the input pool cannot transfer over 8 bytes entropy, $k < 8$. This requirement alleviates the denial of service attack[2] and provides a threshold for backward security[5].

Generator runs the output function in Algorithm 2 to extract k bytes from the input pool. Then the k bytes are added into the blocking pool using mix function in Algorithm 1. The entropy counter of the input pool h_i decrements by $8k$ bits, while the entropy counter of the blocking pool h_o is increased by $8k$ bits. If the new h_o equals or greater than $8k$, the generator outputs k bytes random data from blocking pool. The entropy counter of the blocking pool decreases k bytes. Otherwise, output generation in block pool stops after $\lfloor h_o/8 \rfloor$ bytes, and only continues when enough entropy is mixed into the input pool.

Algorithm 3 represents the above nature language description.

Algorithm 3 $blockingPoolGeneration(pool, k)$

```
Input: block pool $pool$, input pool $inpool$, $h_i$, $h_o$, k.
Output: generating k bytes random data
1: if $h_o < 8k$ then
2:     $k^{'} \leftarrow min(max(k - \lfloor h_o/8 \rfloor, 8), 128)$
3:     $k^{'} \leftarrow min(k^{'}, \lfloor h_i/8 \rfloor)$
4:     if $k^{'} \geq 8$ then
5:         $trans \leftarrow byte[k^{'}]$
6:         $trans \leftarrow output(inpool, k^{'})$
7:         $mix(pool, trans)$
8:         $h_o \leftarrow h_o + 8k^{'}$
9:         $h_i \leftarrow h_i - 8k^{'}$
10: if $h_o < 8k$ then
11:     $output(pool, \lfloor h_o/8 \rfloor)$
12:     $h_o \leftarrow h_o - 8\lfloor h_o/8 \rfloor)$
13:     while input pool has not enough entropy do
14:         Wait
15:     $blockingPoolGeneration(pool, k - \lfloor h_o/8 \rfloor)$
16: else
17:     $output(pool, k)$
18:     $h_o \leftarrow h_o - 8k$
```

Generating data from nonblocking pool. k bytes random data are requested to generate by /dev/urandom. Let h_i be the current entropy counter of the input pool, h_o be the current entropy counter of the nonblocking pool. If $h_O < 8k$, a transfer request is sent to the input pool to transfer $k - \lfloor h_o/8 \rfloor$ bytes entropy (usually$\lfloor h_o/8 \rfloor = 0$). Then input pool extracts k bytes, $k = min(\lfloor h_1/8 \rfloor - 16,\ k - \lfloor h_o/8 \rfloor)$. This means that the input pool leaves at least 16 bytes entropy in the pool and never transfers more bits than its entropy counter allows. Moreover, no transfer is done when the input pool cannot transfer more than 8 bytes entropy, $k < 8$.

Generator runs the output function in Algorithm 2 to extract k bytes from the input pool. Then the k bytes are added into the nonblocking pool using mix function in Algorithm 1. The entropy counter of the input pool h_i decrements by $8k$ bits, while the entropy counter of the nonblocking pool h_o is increased by $8k$ bits. In contrast to block pool, random bits are generated from non blocking pool until k bytes are generated, regardless of whether input pool has enough entropy to transfer as requested.

The above nature language description is represented in Algorithm 4.

Algorithm 4 $nonblockingPoolGeneration(pool, k)$

```
Input: nonblock pool $pool$, input pool $inpool$, $h_i$, $h_o$, k.
Output: generating k bytes random data
1: if $h_o < 8k$ then
2:     $k^{'} \leftarrow min(max(k - \lfloor h_o/8 \rfloor, 8), 128)$
3:     $k^{'} \leftarrow min(k^{'}, \lfloor h_i/8 \rfloor - 16)$
4:     if $k^{'} \geq 8$ then
5:         $trans \leftarrow byte[k^{'}]$
6:         $trans \leftarrow output(inpool, k^{'})$
7:         $mix(pool, trans)$
8:         $h_o \leftarrow h_o + 8k^{'}$
9:         $h_i \leftarrow h_i - 8k^{'}$
10: $output(pool, k)$
11: $h_o \leftarrow h_o - 8k$
```

V. Security Analysis

CSPRNG with entropy inputs must meet several security requirements [5][2]:

Pseudorandomness. An attacker with no knowledge of the internal state of the generator and the entropy input, he can not compromise the internal state and/or predict future outputs from current outputs. Moreover, if the attacker controls the entropy input but has no knowledge of the internal state, he will gain no additional information on the new state and/or future outputs.

Forward security. An attacker with knowledge of the current internal state should be unable to recover the previous outputs of the generator. Forward security can be provided by 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 based on the knowledge of its current internal state. If an adversary knows the internal state, he will predict the corresponding output as well as future outputs until enough entropy is used to refresh the pool since the output function is deterministic.

The internal state of LRNG includes the content of input pool, blocking pool and nonblocking pool. The following sections analyze the LRNG based on the three requirements.

A. Pseudorandomness

1) : An attacker with no knowledge of the internal state of the generator and the entropy input, he can not compromise the internal state and/or predict future outputs from current outputs.

Only with the knowledge of the current output, the attacker can not compromise the content of the output pools and the input pools. Considering the SHA-1 function is one way, the attacker can not recover the corresponding content of the output pool if he only knows the outputs. The folding operation also help to eliminate recognizable patterns in the hash function output. Since the attacker has no control of the entropy input and assuming the output pool is not compromised, he cannot learn anything about the input pool. The input pool has a one-way output, the attacker can not recover the corresponding content of the input pool even if he knows the one-way output. To learn those bits, the attacker at least should know the content of the output pool since the one-way output is mixed into the output. This conflicts with the assumption.

The attacker can not predict the future output since he does not know the three pools and does not know of the entropy input.

2) : An attacker with no knowledge of the internal state of the generator but controls the entropy input. The design of the mixing function guarantees that if an attacker controls the entropy input but does not know of the input pool, he will gain no additional information on the new state after execution of the mixing function.

B. Forward Security

If an attacker has knowledge of both the output pool and the input pool, then he knows the previous state except for the 160 bits (5 words hash value) which were fed back during the last output generation. This leads to a default resistance of 160 bits, the attack overhead is 2^{160}. In the kernel version 2.6.10, only 3 words are fed back to the pool after compression function of the pool. This allowed an overhead of 2^{96} computations, compared to 2^{160}.

C. Backward Security

1) : The attacker knows the output pool, but not the input pool. The attacker will keep his knowledge of the output pool until k bits are transferred from the input pool and mixed into the output pool. The complexity of guessing the k bits is 2^{k-1} in average. No transfer is done when the input pool cannot deliver over 8 bytes of entropy gives an overhead complexity of 2^{64}. If a single byte has been transferred. The backward security is only 8 bits.

2) : An attacker has knowledge of both the output pool and the input pool, But if k bits of entropy are collected before the adversary sees the output, the complexity of guessing the input is still 2^{k-1} on average. This again leads to a default resistance of 64 bits.

VI. Conclusion

The survey narrows down random number generators in cryptographic applications to the study of Linux Random Number Generator. The technical and scientific details of the LRNG are explained and illustrated. The contributions of the survey includes i) generalizing the building blocks in LRNG as algorithms, such as mix function, output function. Both functions are explained with details in the case of 128 words pool and 32 words pool. Especially for the output function, current literature only contains the detailed description of the output function on a 32 words pool. This survey generalizes the pool size to n and comprehensively illustrates how the function works on a n words pool. ii) providing the complete procedure of random data generation for blocking pool and nonblocking pool. By linking different building blocks together, one should have a better understanding of LRNG. iii) summarizing the resilience of LRNG under cryptographic attacks.

This survey studies twoversions of LRNG, namely kernel 2.6.10, kernel 2.6.30.7. Future work may analyse the most recent version of Linux kernel(4.19.12) and compare the difference.

References

[1] M. Alimomeni, R. Safavi-Naini, and S. Sharifian. A true random generator using human gameplay. In *International Conference on Decision and Game Theory for Security*, pages 10–28. Springer, 2013.

[2] Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the linux random number generator. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[3] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[4] R. Kumari, M. Alimomeni, and R. Safavi-Naini. Performance analysis of linux rng in virtualized environments. In *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop*, pages 29–39. ACM, 2015.

[5] P. Lacharme, A. Rock, V. Strubel, and M. Videau. The linux pseudorandom number generator revisited. 2011.

[6] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Booz-Allen and Hamilton Inc Mclean Va, 2001.