PRNGs in Linux

Zane Goldhahn (p72n371), Garrett Perkins (m95m353), Ethan Fison (t85j427), John Dolph (r87f693)

CSCI460 -- Operating Systems
November 15, 2020

Table of Contents

1. Introduction

1.1 Introduction

This technical report covers the implementation of random number generation in Linux from the hardware level up to the cryptographic algorithms used to generate pseudorandom numbers from the entropy pool. Background information on the usefulness of random numbers and a brief overview of Linux random number generation is provided followed by sections for details of random number generation in Linux. These details include the gathering of entropic data for the entropy pool, cryptographic algorithms for generating additional pseudorandom numbers, and hardware-level random number generators. Additionally, the Microsoft Windows implementation of random number generation is investigated for comparison to that of Linux.

2. Background

2.1 Background Information

Random numbers are widely used in modern computer systems. Applications include computer games, cryptographic keys such as RSA keys[8], and Monte Carlo simulations(often used in financial markets)[9]. Of particular relevance is the use of random numbers in cryptography due to its use in computer security. To generate keys suitable for encryption there must be sufficient randomness so that it is impractical for a would be attacker to launch a brute force attack. This requires collecting entropy from sources that are unpredictable and controlling access to the entropy pool. Modern operating systems provide random number generators capable of meeting these demands.

The Linux kernel includes a random number generator that maintains a pool of up to 4096 bits of entropy. This entropy pool is replenished with entropy from external sources and the estimated amount of available entropy is tracked. Care is taken such that when numbers generated from the pool are exposed, the contents of the pool and previous/future randomly generated numbers are not compromised. The user interface is provided by the special files /dev/random and /dev/urandom (present since Linux 1.3.30) [4]. Figure A.1 shows how all of this fits together. Further details are hashed out in the body of this report. Additional files containing information such as pool size(4096 bits since Linux 2.6) and the estimated amount of available entropy are located in /proc/sys/kernel/random [4].

3. Gathering Entropic Data

3.1 Sources of Entropy

Computers are deterministic and therefore are generally not good sources of entropy. This necessitates gathering entropy from external sources, often the user of the system. These sources include the timing of events caused by the user, such as key presses and mouse movements. It is possible, however, that the user may not generate enough entropy to keep the entropy pool filled to a safe level. This could occur if the user is away or if the system does not interact directly with the user; for example, a server. To address this, entropy is collected from other sources as well. These include timing information from interrupts and disk I/O. However, due to the increasing prevalence of SSDs with their much less variable I/O times, disk I/O is becoming a less reliable source of entropy. Additionally, some processors include a hardware random number generator which is used as an entropy source if available. The use of multiple entropy sources both reduces the probability of the entropy pool being depleted, and increases the number of entropy sources an attacker would need to control to compromise the entropy pool.

3.2 Entropy Collection

Entropy input to the entropy pool is handled by functions located between lines 1100 and 1332 in the `random.c` file [1]. There is an entropy bit count which keeps track of the amount of entropy in the pool and this count is updated whenever entropy is contributed. The following four functions are used to gather entropy:

```
void add_device_randomness(const void *buf, unsigned int size);
void add_interrupt_randomness(int irq, int irq_flags);
void add_input_randomness(unsigned int type, unsigned int code, unsigned int
value);
void add_disk_randomness(struct gendisk *disk);
```

`add_device_randomness` does not actually add any entropy. Rather, it addresses the issue that identical devices are likely to have very similar initial states by incorporating data specific to each device such as MAC addresses and serial numbers into the entropy pool. This makes it more difficult to crack

`add_interrupt_randomness` adds entropy from irq timings at a rate of roughly once per second [1]. Its inputs are the cycle counters and irq source. This function uses exclusive OR to combine the cycles and irq into the `fast_pool` before calling `__mix_pool_bytes` to mix the `fast_pool` into the entropy pool. If an architectural seed generator exists this is also where a seed from it will be incorporated into the entropy pool. Upon completion this function credits the entropy bit count with one bit unless an architectural seed generator exists in which case two bits are credited.

The next two functions add_input_randomness and add_disk_randomness are used to collect entropy from user input and disk I/O. Simple checks are made to avoid auto repeat keypresses and such. While the introduction of similar events will not necessarily reduce the entropy in the pool(thanks to the mixing function) it is likely to result in an overestimation of available entropy and so is avoided. Both of these functions call add_timer_randomness which calls _mix_pool_bytes to mix the new entropy into the entropy pool. Bits are then credited to the entropy bit count. The number of bits credited depends on the estimated amount of entropy contributed but will not exceed 12. This number is estimated based on the first, second, and third order deltas as shown in the following snippet from the Linux source code.

```
delta = sample.jiffies - READ_ONCE(state->last_time);
WRITE_ONCE(state->last_time, sample.jiffies);

delta2 = delta - READ_ONCE(state->last_delta);
WRITE_ONCE(state->last_delta, delta);

delta3 = delta2 - READ_ONCE(state->last_delta2);
WRITE_ONCE(state->last_delta2, delta2);

if (delta < 0)
      delta = -delta;
if (delta2 < 0)
      delta2 = -delta2;
if (delta3 < 0)
      delta3 = -delta3;
if (delta > delta2)
      delta = delta2;
if (delta > delta3)
      delta = delta3;
```

Source [1].

This code calculates the first three deltas and then finds the minimum delta which is later rounded down by one and added to the entropy counter. Calculating the entropy added in this way prevents the available entropy from being overestimated when similar events contribute to the entropy pool in a short period of time.

3.3 Entropy Pool Carryover

At startup it is possible to have very little entropy available due to the predictable manner in which the system starts. This could be particularly troublesome for a remote system which may need to establish network connections before interacting with a user. Without sufficient entropy, the generated network encryption keys would be weak, leaving the system vulnerable to attack. To prevent this a seed file can be saved across reboots as follows.
In shutdown script:

```
# Carry a random seed from shut-down to start-up
# Save the whole entropy pool
echo "Saving random seed..."
random_seed=/var/run/random-seed
touch $random_seed
chmod 600 $random_seed
poolfile=/proc/sys/kernel/random/poolsize
[ -r $poolfile ] && bits=$(cat $poolfile) || bits=4096
bytes=$(expr $bits / 8)
dd if=/dev/urandom of=$random_seed count=1 bs=$bytes
```

Source: [4]

In startup script:

```
echo "Initializing random number generator..."
random_seed=/var/run/random-seed
# Carry a random seed from start-up to start-up
# Load and then save the whole entropy pool
if [ -f $random_seed ]; then
      cat $random_seed >/dev/urandom
else
      touch $random_seed
fi
chmod 600 $random_seed
poolfile=/proc/sys/kernel/random/poolsize
[ -r $poolfile ] && bits=$(cat $poolfile) || bits=4096
bytes=$(expr $bits / 8)
dd if=/dev/urandom of=$random_seed count=1 bs=$bytes
```

Source: [4]

This saves the contents of the entropy pool at shutdown and then reloads the entropy pool with these contents when the system restarts. This addresses the problem of insufficient entropy at startup and provides a cryptographically secure output as soon as it is loaded. It has been included in all major linux distributions since at least 2000 [4].

4. Cryptographic Algorithms

4.1 SHA-1 Hash Algorithm

The purpose of a cryptographic hash function is to generate a fixed size, irreversible numerical output based on a variable-size input. An irreversible output means that the original value inserted into the hash function will not be obtainable by the output value.

Each value produced by the SHA-1 hash function is a 160-bit hash value rendered as a 40 digit hexadecimal number. The rendered value is referred to as a “message digest.” A string-valued message that is to be operated on will first be encoded into its binary representation based on each character’s ASCII code. Each character binary representation must be 8 bits. We may pad 0-valued bits onto the most-significant bit until this condition is true. The end of the message encoding (starting at the least-significant bit) will have a 1-valued bit padded to it. 0-valued bits will then be appended until the binary message has a length of 512 modulo 448 bits. A 64-bit representation of the message length will then be padded to the end of the message. We may convert the length integer value to binary and pad 0-valued bits onto the most-significant bit until there are 64 bits. The result of these operations will then be a multiple of 512 bits, which will then be broken down into separate 512-bit chunks. Each chunk will then be divided into sixteen 32-bit big-endian pieces. Once we have divided our 512-bit chunks of data into 32-bit pieces, we will extend each chunk’s collection of 32 bit pieces to a collection of 80 pieces. The procedure is to iterate 80 times in a for loop and compute the following formula for iterations 16-79 (the number 80 is fixed for the SHA-1 algorithm). Each w refers to one of the 32-bit pieces identified by its index. The value i is the current iteration:

```
w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrotate 1
```

The *leftrotate* operation is a bitwise left circular shift performed on the value *w[i]*. In a left circular shift, every bit is shifted one place towards the most-significant bit position and the most significant bit takes on the least-significant bit position.

Prior to performing the above operations on the original binary encoded message, five random hex values were generated. We may refer to these values as h0, h1, h2, h3, and h4. Following all of the operations outlined above, we will iterate 80 times in a second for loop. The next step will be to iterate through each chunk while performing bitwise functions and variable reassignments. In particular, h0, h1, h2, h3, and h4 will be reassigned according to the following bitwise operations:

```
    Initialize hash value for this chunk:
     a = h0
     b = h1
     c = h2
     d = h3
     e = h4

     for i from 0 to 79
         if 0 ≤ i ≤ 19 then
             f = (b and c) or ((not b) and d)
             k = 0x5A827999
         else if 20 ≤ i ≤ 39
             f = b xor c xor d
             k = 0x6ED9EBA1
         else if 40 ≤ i ≤ 59
             f = (b and c) or (b and d) or (c and d)
             k = 0x8F1BBCDC
         else if 60 ≤ i ≤ 79
             f = b xor c xor d
             K = 0xCA62C1D6

         temp = (a leftrotate 5) + f + e + k + w[i]
         e = d
         d = c
         c = b leftrotate 30
         b = a
         a = temp

     Add this chunk's hash to result so far:
     h0 = h0 + a
     h1 = h1 + b
     h2 = h2 + c
     h3 = h3 + d
     h4 = h4 + e

 Produce the final hash value (big-endian) as a 160-bit number:
 hh = (h0 leftshift 128) or (h1 leftshift 96) or (h2 leftshift 64) or (h3 leftshift
 32) or h4
```

The final variable value, `hh`, is the message digest. This is the final value that the SHA-1 hash algorithm is intended to produce.

Here is an example operation of how the SHA-1 procedure will begin processing a hash value for an arbitrary string value:

- Arbitrary String: **xyz**
- Convert to Binary: **01111000 01111001 01111010**
- Create Random Hex Values (these hash values are provided as a particular example):

  h0 = 0x67452301
  h1 = 0xefcdab89
  h2 = 0x98badcfe
  h3 = 0x10325476
  h4 = 0xc3d2e1f0

- Pad Message: **01111000 01111001 01111010 1000000000 ... (448 bits)**
- Add length of message represented as 64 bits:

  Bit length = **24 bits**
  24 converted to binary = **11000**
  Binary after padding: = **0000 ... 11000 (64 bits)**

- Message : **01111000 01111001 01111010 10000 ... 11000 (512 bits)**
- Break the message into sixteen 32-bit words:

[01 1 1 1 000 0 1 1 1 1 001 01 1 1 1 010 10000000,
00000000 00000000 00000000 00000000,
. . .
00000000 00000000 00000000 000 1 1 000] (Collection of sixteen 32-bit words)

- Extend the collection of 32-bit pieces to a collection of 80 32-bit pieces and reassign the values for h0 - h4 as defined in the pseudocode above. I have left this operation explanation to the pseudocode since it is a lengthy process.

- An SHA-1 hash is: **9209ca056381a67511973799b9200e5de5b40d1c**
  For the string: **xyz**

The SHA-1 hash function is used within the Linux PRNG output function (entropy extraction algorithm). The output function is a non-linear operation for either transferring entropy between the entropy input and output pools, or generating data from an entropy output pool. Every time SHA-1 generates a 160-bit hash value, the system entropy estimate will be reduced by 160 bits accordingly. Although the exact details of SHA-1 are not pertinent for understanding the Linux pseudorandom number generator at a high level, I included a detailed analysis for completeness of this research project. The implementation of the SHA-1 function is resident in the Linux crypto/sha1_generic.c file. The implementation in Linux is the same as described above.

The above explanations and formulas were derived from Wikipedia information. The pseudocode chunks, except for the arbitrary string example, were clipped from Wikipedia. See [3] for the full SHA-1 pseudocode and [2] for its implementation in Linux.

4.2 Entropy Pool Mixing Algorithm

The purpose of the entropy pool mixing algorithm is to ensure randomness. Mixing multiple sources of low entropy will yield a source containing greater entropy. Mixing the entropy pool will prevent a potential attacker from using any insight that they may have in regard to how the entropy was gathered.

The Linux entropy mixing function operates on one byte at a time, extending each byte into a 32-bit word. The 32-bit word will then be rotated and shifted back into the entropy pool by a linear shift register. All entropy will be accounted for during the mixing process and will re-enter the entropy pool. The mixing function is depicted in the following diagram:

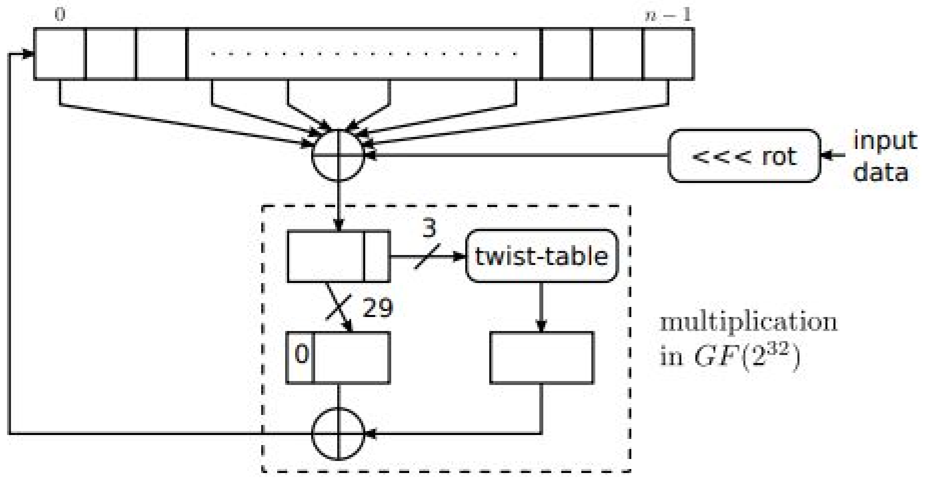


Figure 3: The mixing function.

Prior to being shifted back into the entropy pool, the 32-bit value will undergo multiplication in GF(2^32). GF refers to finite field arithmetic, which deals with a finite number of elements as opposed to an infinite number of elements. The finite field arithmetic operation is depicted in the diagram above as labeled. For the purpose of this paper, I will provide a concise, non-theoretical explanation of the mixing operations. Each byte will be rotated by the rol32 function and stored as the variable w. The taps in the pseudocode below refer to sections of the entropy pool to which the rotated value, w, will be XOR with. The arrows extending from the array (entropy pool) in the diagram above refer to the tap locations. This first XOR operation is depicted by the upper circle in the diagram. The circles in the diagram refer to the bitwise XOR operations. The last 3 bits of the 32-bit value will be used to look up a value in the twist table, and the first 29 bits of the 32-bit word will be padded at the left with 0-valued bits. The twist-table is an array of 32-bit hexadecimal values. One of these values will always be selected. The twist-table is defined as follows in the Linux source code:

```
static __u32 const twist_table[8] = {
        0x00000000, 0x3b6e20c8, 0x76dc4190, 0x4db26158,
        0xedb88320, 0xd6d6a3e8, 0x9b64c2b0, 0xa00ae278 };
```

The padded value will be XOR with the selected twist-table hexadecimal value. The entire operation is outlined in the Linux source code below. The variable nbytes refers to the number of bytes waiting to be mixed:

```
static void _mix_pool_bytes(struct entropy_store *r, const void *in,
                            int nbytes)
{
        unsigned long i, tap1, tap2, tap3, tap4, tap5;
        int input_rotate;
        int wordmask = r->poolinfo->poolwords - 1;
        const char *bytes = in;
        __u32 w;

        tap1 = r->poolinfo->tap1;
        tap2 = r->poolinfo->tap2;
        tap3 = r->poolinfo->tap3;
        tap4 = r->poolinfo->tap4;
        tap5 = r->poolinfo->tap5;

        input_rotate = r->input_rotate;
        i = r->add_ptr;

        /* mix one byte at a time to simplify size handling and churn faster */
        while (nbytes--) {
                w = rol32(*bytes++, input_rotate);
                i = (i - 1) & wordmask;

                /* XOR in the various taps */
                w ^= r->pool[i];
                w ^= r->pool[(i + tap1) & wordmask];
                w ^= r->pool[(i + tap2) & wordmask];
                w ^= r->pool[(i + tap3) & wordmask];
                w ^= r->pool[(i + tap4) & wordmask];
                w ^= r->pool[(i + tap5) & wordmask];

                /* Mix the result back in with a twist */
                r->pool[i] = (w >> 3) ^ twist_table[w & 7];

                /*
                 * Normally, we add 7 bits of rotation to the pool.
                 * At the beginning of the pool, add an extra 7 bits
                 * rotation, so that successive passes spread the
                 * input bits across the pool evenly.
                 */
                input_rotate = (input_rotate + (i ? 7 : 14)) & 31;
        }

        r->input_rotate = input_rotate;
        r->add_ptr = i;
}
```

The mixed entropy will be moved back into whichever entropy pool is connected to the mixing function output. See [1] for the origin of the above Linux source code. The above code is the _mix_pool_bytes function.

Calling the function _mix_pool_bytes located in the drivers/char/random.c file will begin the mixing procedure. The _mix_pool_bytes function begins at line 537. All of the entropy mixing algorithms are implemented in the drivers/char/random.c file beginning at line 537. See [1].

See [1] for the origin of the above pseudocode, and [5] for the original source of the above diagram.

4.3 Entropy Extraction Algorithm

For pseudorandom number generation in Linux, an SHA-1 hash of the entire initial entropy output pool is taken. The hash value is then used as an initial value for a second SHA-1 hash value, and is also mixed back into the output pool by the entropy pool mixing algorithm. The value of the first hash and 16 additional words (64 additional bytes) will be used as inputs to a second SHA-1 hash. Once the new hash is generated, it will be folded in half to produce a 10 byte output. The folding operation will be performed as in the following pseudocode from Linux. See [1] line 1434. The w values refer to 4 byte words of the second SHA-1 hash value where the index refers to the word position relative to the other words. The third code line means that bits 0 through 15 of the second word will be XOR with bits 16 through 31 of the second word:

```
        /*
         * In case the hash function has some recognizable output
         * pattern, we fold it in half. Thus, we always feed back
         * twice as much data as we output.
         */
        hash.w[0] ^= hash.w[3];
        hash.w[1] ^= hash.w[4];
        hash.w[2] ^= rol32(hash.w[2], 16);
```

The folding operation output is the final value of the entropy extraction process. The outputs are stored in the /dev/random and /dev/urandom directories. /dev/random will block a process requesting a pseudorandom output when there is not enough entropy to produce an unpredictable output. /dev/urandom is non-blocking and will always return an output. Since there is no guarantee on the cryptographic security of /dev/urandom, /dev/random is better suited for dealing with highly sensitive data since it guarantees a substantial amount of entropy was used to compute its contents.

The entropy pool operations to produce the final pseudorandom outputs stored in /dev/random and /dev/urandom are depicted in the following diagram:

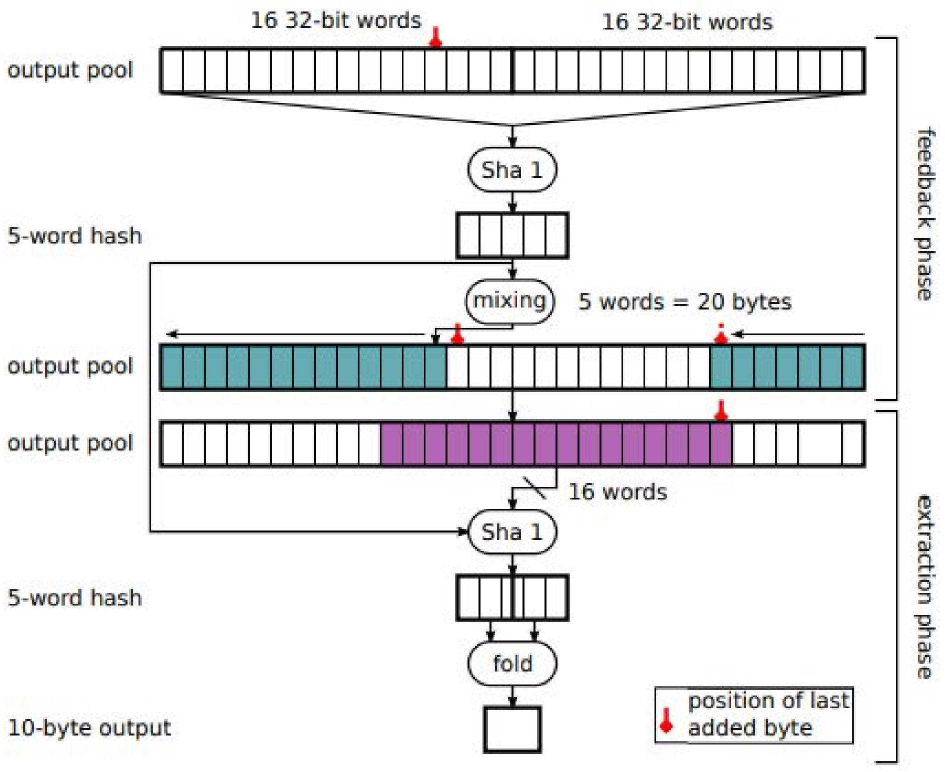


Figure 5: The output function of the Linux PRNG.

The entropy extraction algorithms are implemented in the drivers/char/random.c file beginning at line 1335. See [1]. *See [5] for the original source of the above diagram.*

5. Hardware-Level PRNGs

5.1 Overview of HRNGs

Hardware random number generators (HRNGs) or true random number generators (TRNGs) are devices that generate random numbers from a physical process rather than from an algorithm. There are many different options when it comes to HRNG. Many of the devices are based on microscopic phenomena that generate low-level, random noise signals. Examples include, thermal noise, radioactive decay, the photoelectric effect and other quantum phenomena. In theory all of these processes are completely unpredictable. Also, since the HRNGs have a limited bit rate, they are often used as the seed for a faster PRNG, which can output a much higher data rate. HRNGs are mainly used for cryptography to generate cryptographic keys to transmit data securely. A common example of this is Internet encryption protocols such as Transport Layer Security (TLS).

While Linux PRNG is intended to be usable without any specific hardware-level PRNGs, it does include a number of drivers for hardware RNGs. The outputs from these generators, however, are not mixed into the Linux PRNG and are accessed through /dev/hwrng.

HRNGs do have some negative publicity. There are numerous articles that talk about the idea of backdooring HRNGs and also that they may not always generate random numbers. Some of this is due to the use of observed events such as the time between keystrokes, or other external events. This can cause problems if a malicious attacker can control these external inputs. This would destroy the randomness used for cryptography and make the system vulnerable. There can also be failure of the hardware generating the random numbers. Most of the time it will fail silently and give no indication that the output is no longer random. It is important when using HRNGs to have tests running to confirm that the output is indeed random.

5.2 Linear Feedback Shift Registers

LFSRs (linear feedback shift registers) provide a simple means for generating non sequential lists of numbers quickly on microcontrollers. They can be implemented on both hardware and software levels. Generating the pseudo-random numbers only requires a right-shift operation and an XOR operation[10]. They can be implemented across a number of hardware platforms such a C and Assembly. Later in this section there will be examples of how they are implemented in these languages.

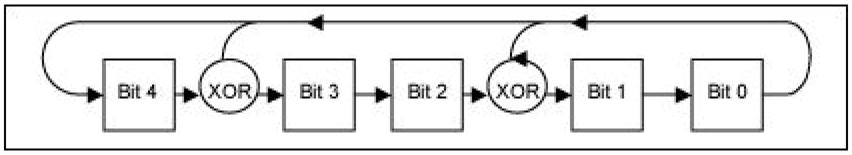
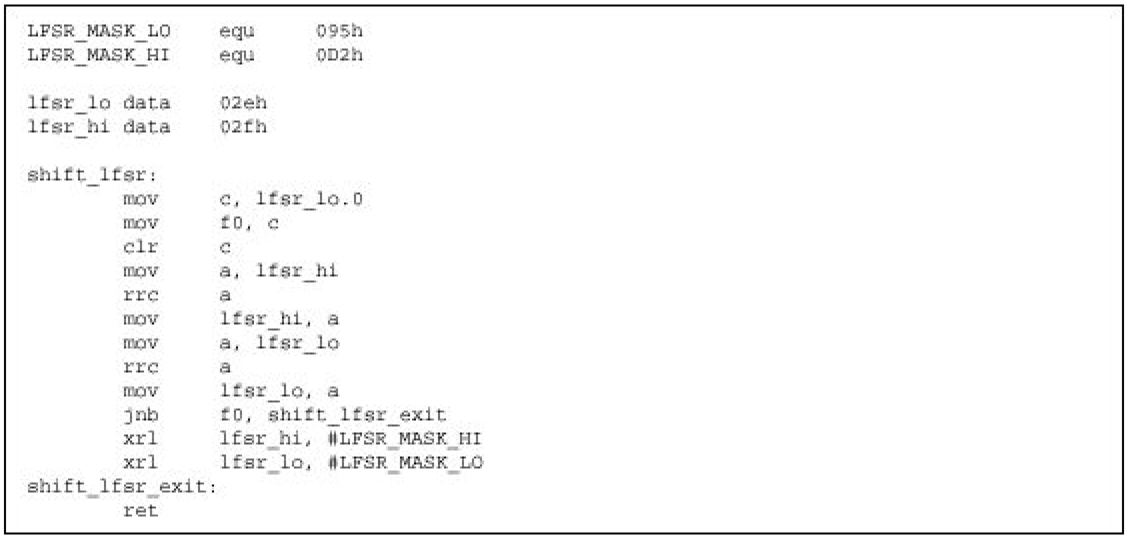


Figure 6: A simplified drawing of a LFSR[10]

LFSRs are entirely specified by their polynomials, and can never have a zero value because they are seeded to a nonzero value. For example, a 5th-degree polynomial with every term present is represented with the equation $x^5 + x^4 + x^3 + x^2 + x + 1$ [10]. There are $2^{(5-1)} = 16$ different possible polynomials of this size. A basic LFSR will not produce a good set of random numbers. There are many ways to increase the suitability of the number produced. First, instead of using a basic LFSR, the size of the LFSR is increased and only the lower bits are used for the random number. For example, "if you have a 10-bit LFSR and want an 8-bit number, you can take the bottom 8 bits of the register for your number"[10]. While this does produce a better result than just a basic LFSR. XORing can also be used to increase the randomness of LFSRs by XORing one LFSR with another of a different size, but this must be done with caution because the LFSR could become a zero term.

Below, in Figure 7, we see the implementation of a RNG LFSR using assembly. A is a register and c is the carry register. The *mov* function is used to move the bits around, and then they are rotated right with a carry (*rrc*). This is the shift part of the program. At the very end *xrl* is used to XOR the hi and low.



```
LFSR_MASK_LO    equu     095h
LFSR_MASK_HI    equu     0D2h

lfsr_lo data    02eh
lfsr_hi data    02fh

shift_lfsr:
        mov     c, lfsr_lo.0
        mov     f0, c
        clr     c
        mov     a, lfsr_hi
        rrc     a
        mov     lfsr_hi, a
        mov     a, lfsr_lo
        rrc     a
        mov     lfsr_lo, a
        jnb     f0, shift_lfsr_exit
        xrl     lfsr_hi, #LFSR_MASK_HI
        xrl     lfsr_lo, #LFSR_MASK_LO
shift_lfsr_exit:
        ret
```

Figure 7: Assembly code to implement a 16-bit LFSR[10]

LFSRs have long been used as PRNGs for use in stream ciphers, but because it is a linear system it does have some flaws. This is fixed by adding nonlinear combinations of several bits from the LFSRs, or irregular clocking of the LFSR. Some notable uses for LFSRs are Bluetooth, GSM cell phones, and E0.

5.3 Other Hardware-Level PRNGs

On Chip HRNGs

RDRAND for read random, known as Intel Secure Key Technology, is an instruction set for returning random numbers from an Intel on-chip HRNG. This has been seeded by an on-chip entropy source. Both Intel and AMD, the two main manufacturers of CPUs and other computer chips have built in HRNGs that make use of RDRAND and RDSEED x86 instructions.

Intel has developed a digital random number generator (DRNG) using underlying DRNG hardware implementation. The DRNG follows the same cascading model as a RNG by using an entropy source inside the processor that repeatedly seeds a hardware CSPRNG. This allows the entropy source to be sampled quickly and produce high quality entropy. The response times of Intel's DRNG are comparable to other, software level, PRNGs.

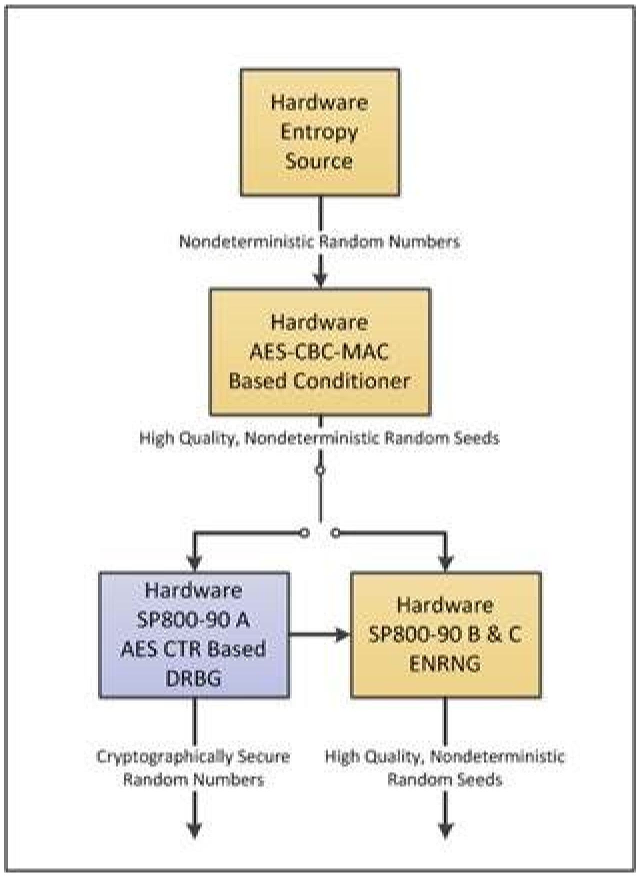

Hardware Entropy Source
Nondeterministic Random Numbers
Hardware AES-CBC-MAC Based Conditioner
High Quality, Nondeterministic Random Seeds
Hardware SP800-90 A AES CTR Based DRBG
Hardware SP800-90 B & C ENRNG
Cryptographically Secure Random Numbers
High Quality, Nondeterministic Random Seeds

Figure 8: Digital Random Number Generator component architecture [19]

The components of the DRNG form an asynchronous pipeline: "an entropy source (ES) that produces random bits from a nondeterministic hardware process at around 3 Gbps, a conditioner that uses AES (4) in CBC-MAC (5) mode to distill the entropy into high-quality non deterministic random numbers, and two parallel outputs" [19]. Since the container does not send the same value to the DRBG and ENRNG, there is no chance that the software application can obtain the value seed. This is similar to the entropy pool in Linux, but since it is fed by a constant stream of entropy faster than the downstream components can consume the data, there is no need for a pool. Insead there is always fresh entropy. The entropy source for the DRNG is the thermal noise within the silicon. Built-in, self tests, help to maintain the functionality of the DRNG [19].

FPGA RNGs

FPGA or Field Programmable Gate Arrays can be used to create many different types of RNGs as they are reconfigurable hardware systems. They allow the user to create a number of hardware solutions, and then to choose the best option for a RNG. These can be classified as linear and nonlinear pseudo and truly random number generators. While these are on an FPGA, the linear and nonlinear options are considered PRNGs. These are not discussed, because they are PRNGs and not TRNGs.

TRNGs

The true random number generators that are created using an FPGA are physical generators that use various hardware components to produce random numbers. This is typically faster than using software. TRNGs use an entropy source such as electronic noise of embedded components, or sensor inputs for temperature, noise, etc. This makes FPGAs an inexpensive and efficient option for generating random numbers.

There are a few types of TRNGs that can be implemented on an FPGA. A Phase-Locked Loop, a Ring Oscillator, a Self-Timed Ring, and a Metastability ring. The only one discussed in this paper will be a Phase-Locked Loop.

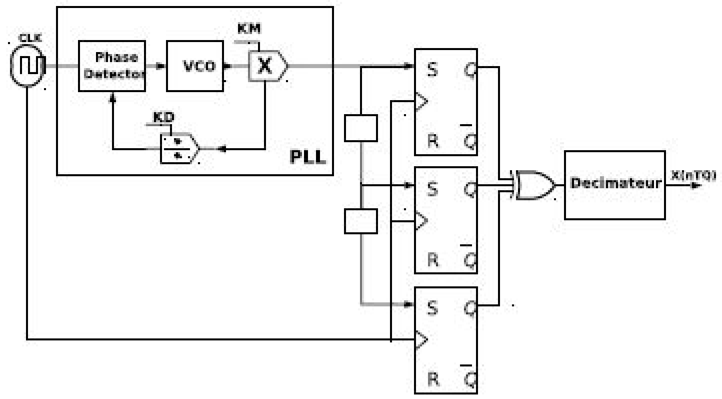


Figure 9: Architecture of a Phase-Locked Loop TRNG [18]

Phase-Locked Loops are derived from an external clock generator source, which can be configured to produce a signal whose phase is associated to the phase of the input signal. The input signal is dependent on factors in the physical environment of the FPGA such as power, temperature, etc. The input signal uses the jitter extraction technique as a random stream of inputs, which is a short-term variation of the clock propagation [18]. If there is no jitter, then the output can be predicted, so to rectify this PLLs are used in either a cascading configuration or a parallel configuration to increase the jitter of the system. This does increase the sensitivity, but not enough to affect the system.

6. Microsoft Windows Implementation

6.1 The Basics

In October of 2019, Microsoft published a whitepaper on the implementation of PRNG in the Windows kernel [11]. Unfortunately, it is rather limited in its level of detail, though we can still use the information available to compare to the Linux implementation.

At its very base, Windows uses the SP800-90 AES_CTR_DRBG cryptographic PRNG for its generation of random numbers (referred to as WinRNG hereafter). There are three major components of this PRNG to make the of. The first, SP800-90, refers to NIST Special Publication 800-90, which provides definitions for NIST-approved RNG mechanisms.
The next component of note is AES. This of course indicates that WinRNG uses AES hashing as part of its process to generate 128-bit output blocks.
The final component is CTR_DRBG, a Deterministic Random Bit Generator in counter mode. This mechanism is defined from pages 48 to 61 of SP800-90 [12].

For requests that ask for less than 128 random bytes, the Windows kernel maintains a buffer that refills whenever emptied. This is primarily a measure to improve performance for smaller requests, though there is a drawback to this: the buffer only refills when empty, so there may be a performance hit if many requests are made in rapid succession. As a mechanism to increase PRNG performance, the Windows kernel maintains separate buffers for each logical CPU core. Note that *logical* includes the additional "cores" of Intel's hyperthreading or AMD's Clustered Multi Threading features in CPUs. In kernel mode, there is only a single buffer per core. However, user mode maintains a much larger set of PRNG buffers - one per core, per process. Additionally, the kernel has a root PRNG, which is used to generate seeds for all other PRNGs in the system. [11]

6.2 Available APIs

As with the previous section, we are fairly limited in what we can glean from the information available. Microsoft does offer a developer site for API documentation similar to that of Oracle's Java documentation, but as Windows and its kernel are closed-source, there is not much information beyond that which is readily available to the general public.

The Windows 10 API exposes five functions for RNG. Following are the function prototypes from the official documentation:
(Function information and prototypes from docs.microsoft.com)

SystemPrng (See [14])

Source: https://docs.microsoft.com/en-us/windows/win32/seccng/systemprng

This function is specific to kernel-level RNG requests.

```
BOOL SystemPrng(
  _Out_ unsigned char pbRandomData,
  _In_  size_t        cbRandomData
);
```

Argument Breakdowns:

- pbRandomData
    - Pointer to the buffer the random number will be stored at
- cbRandomData
    - The number of random bytes to send to the address at pbRandomData

ProcessPrng (See [13])

This function is specific to user-level RNG calls. Information on this function is seemingly unavailable in Microsoft's own documentation, and otherwise only accessible through decompiling a dll and interpreting assembly. However, a GitHub user [13] on the golang repository has provided a "reverse-engineered" version of the code from `bcryptPrimitives.dll` (included below for convenience). We cannot verify the accuracy of this, though the function's arguments do align with what we might expect based on the SystemPrng function.

Reverse-engineered ProcessPrng Code:

```
bool ProcessPrng(void *buf, uint64_t len)
{
  bool ret = true;
  uint64_t bytes_to_read;
  int64_t out;
  uint128_t *aes_key;

  success_1 = 1;
  if (g_rngState == 1)
  {
    ret = false;
    out = 0;
    AesRNGState_select(&aes_key);
    for (; len; len -= bytes_to_read)
    {
      bytes_to_read = 0x10000;
      if (len < 0x10000)
        bytes_to_read = len;
      buf += bytes_to_read;
      ret = AesRNGState_generate(aes_key, buf, bytes_to_read, &out);
    }
  }
  else
  {
    if (g_rngState != 2)
    {
      for (;;)
        *(unsigned int *)0 = 0x78676E72;
    }
    EmergencyRng(buf, len);
  }
  return ret;
}
```

It would seem that this function performs a similar, if not identical role to SystemPrng, though in this case for use in user-mode contexts. Additionally, this function **always** generates some random value, whether through explicit AES generation, or through the EmergencyRng function, the functionality of which we can only speculate. As this is not official code, we cannot say with confidence what EmergencyRng represents within the actual kernel code. Its functionality may be a direct pull from the associated CPU core's buffer without any additional modification, or could possibly even force a reseed of the system's root PRNG, thus causing system-wide regeneration of PRNG buffers. Again, this is only speculation, and not to be taken as fact.

BCryptGenRandom (See [15])

Source: https://docs.microsoft.com/en-us/windows/win32/api/bcrypt/nf-bcrypt-bcryptgenrandom

```
NTSTATUS BCryptGenRandom(
  BCRYPT_ALG_HANDLE hAlgorithm,
  PUCHAR            pbBuffer,
  ULONG             cbBuffer,
  ULONG             dwFlags
);
```

Argument Breakdowns:

- hAlgorithm
    - Pointer to the provider for a hashing algorithm (MD5, SHA-1, SHA245, etc.)

- pbBuffer
    - Pointer to the buffer the random number will be stored at.
- - cbBuffer
    - the bytesize of pbBuffer
- - dwFlags
    - used to change how BCryptGenRandom works. In Windows 10, this argument can take on the following values:
        - 0: No change, function will run using the hashing algorithm specified in the call
        - BCRYPT_USE_SYSTEM_PREFERRED_RNG 0x00000002: Will run the function using the system's preferred hashing algorithm. Notably, this value of dwFlags can only be used when the system's IRQL (Interrupt Request Level) is PASSIVE_LEVEL, the lowest level of interrupt priority in Windows. Under DISPATCH_LEVEL, a pointer to a hashing algorithm must be provided.

CryptGenRandom (See [16])

Source:
https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptgenrandom

According to Microsoft's official online documentation, this function is deprecated. The whitepaper provides further detail that ProcessPrng is now the function used by the Windows cryptographic service providers. The functionality appears to be similar to BCryptGenRandom, with the exception that there is no analog for dwFlags.

```
BOOL CryptGenRandom(
  HCRYPTPROV hProv,
  DWORD      dwLen,
  BYTE       *pbBuffer
);
```

RtlGenRandom (See [17])

Source: https://docs.microsoft.com/en-us/windows/win32/api/ntsecapi/nf-ntsecapi-rtlgenrandom

Official documentation seems to indicate that this API is also deprecated, to be replaced by CryptGenRandom

```
BOOLEAN RtlGenRandom(
  PVOID RandomBuffer,
  ULONG RandomBufferLength
);
```

Argument Breakdowns:

- RandomBuffer
    - The pointer to the buffer where the random number will be stored
- RandomBufferLength
    - The size of the buffer pointed to by RandomBuffer

6.3 Entropy in Windows

Windows maintains a set of pools of entropic data to be used for seeding the system's root PRNG. At boot, only a single pool exists in order to build up sufficient entropy, and is later supplemented by additional pools. Entropic data is spread evenly among the pools. The contents of these pools are not directly used for seeding the root PRNG. Instead, the contents of a pool are hashed through SHA-512, then used for seeding. [11]

Windows reseeds on an increasing interval of triple the previous delay, until reaching a predefined upper limit. At the time the whitepaper was published, that upper limit was one hour. Once the upper limit of the interval is reached, the system then enables the generation of additional entropy pools. These additional pools are generated every time the previously created pool will have been used in reseeding three times. From the whitepaper:

"A new pool is created whenever it would have been used in a reseed if it had existed. Thus, once the reseed interval hits the maximum, there are two more reseeds with just one pool. On the third reseed a second pool would have been used, so it is created. From that point forward entropy events are divided between the pools, and pool 1 is used every third reseed. Similarly, 9 reseeds later, the third pool is enabled" [11], page 7

As more pools are generated, the system will use the outputs from multiple pools at once for seeding. From the whitepaper:

"The N pools are numbered 0,…,N-1. Pool 0 is used on every reseed. Pool 1 is used (in addition to pool 0) every 3rd reseed, pool 2 is used every 9th reseed (in addition to pools 0 and 1), etc. In general, pool k is used every 3kth reseed (in addition to all the lower numbered pools)." [11], page 7

Beyond this, no further detail on how multiple pools are used is given. The final output from these combined pools may be the result of further hashing of the pool outputs, or perhaps a combination of the pools prior to hashing.

7. Conclusion & Future Direction

7.1 The Importance of Cryptographic Algorithms

Cryptographic algorithms provide a method of generating useful data from sources of entropy, while ensuring that the generated data is unpredictable and irreversible to its original input. The SHA-1 hash algorithm generates an irreversible value from entropy pool data. Since the original input may not easily be restored, we have introduced a characteristic of randomness. This characteristic is that randomness may not be traced back to any specific initial states. The entropy pool mixing algorithm provides a method of ensuring that the entropy pool has a high amount of entropy. With low entropy, an attacker may possess some insights related to the particular data processed. With randomness, there should not be any predictable internal state for the system in which the randomness is from. The entropy extraction algorithm simply provides a mechanism to deliver random data for practical use by applications.

7.2 Conclusion

The focus of this report was to examine four major topics within the field of pseudorandom number generation: the methods used by the Linux kernel to gather entropic data for use in random number generation, the cryptographic functions used by the Linux kernel for random number generation, hardware-level implementations of random number generation, and a brief overview of random number generation within the Windows 10 kernel. For entropy, we found that Linux has four primary mechanisms for adding entropic data to the pool. These mechanisms take data such as hardware-specific identifiers, interrupt timings, variances from human interface devices such as mice and keyboards, and disk I/O times. Additionally, we noted that the Linux kernel saves the entropy pool between boots as a security feature to ensure that the system always has enough entropic data.

The cryptographic algorithms section is intended to demonstrate how the Linux kernel produces random numbers from the collected entropy. The various sections cover, in detail, the processes of the SHA-1 Hash Algorithm and how it ensures that no input can be gleaned from the output value. There should be no possibility for randomness to be traced back to initial states. Proceeding then to the algorithm that mixes the “pool” to ensure high entropy is always present, and finally, finishing with a look into the extraction algorithm, and how the final outputs are produced for use by applications within the operating system. This section is the main focus of the paper, because it directly pertains to the Linux PRNG.

Even though there is no need for a hardware random number generator in Linux, there is support for them in the Linux kernel, and overall they are important to cryptography. This is a huge topic in it of itself and there was not enough time to touch on every aspect of HRNGs. However, the topics that were covered help to glean insight into the world of HRNGs. The overview of HRNGs helps show how they differ from PRNGs and the advantages and disadvantages of them. Then the section of LFSRs shows how PRNGs can be implemented on the most basic of hardware levels and languages. It was also noted that computer CPUs have integrated RNGs. The FPGA is one of the most popular pieces of hardware used to make HRNGs, and one type of HRNG was discussed.

Finally, for the sake of comparison to Linux, we investigated the implementation of PRNGs within the Windows kernel. Due to the closed-source nature of Windows, we were unable to directly examine any code as we were with Linux. However, through investigating a whitepaper published by Microsoft and an NIST publication regarding random bit generation algorithms, we found that Windows has a rather complex system for seeding its PRNGs, utilizing a single root PRNG to seed all others, and multiple entropy pools to further increase the unpredictability of its seeds.

7.3 Future Direction

This report has covered the implementation of random number generation in Linux. Now that the implementation is understood a logical next step would be to verify the randomness provided first hand. To do this tests could be run to look for any patterns in the output of the random number generator. NIST provides a test suite to verify the suitability of a random or pseudo random number generator for cryptographic applications [6]. This includes frequency tests, runs tests, and many more. An interesting follow-up would be to implement these tests, likely in Matlab or a similar program, and verify the efficacy of random number generation in Linux under various circumstances, particularly soon after boot and in a system with minimal access to user generated entropy.

8. References

[1] *"Linux source code,"*Accessed on: Nov. 5, 2020. [Online]. Available: https://github.com/torvalds/linux/blob/master/drivers/char/random.c

[2] *"Linux SHA-1 Implementation,"*Accessed on: Nov. 5, 2020. [Online]. Available: https://github.com/torvalds/linux/blob/master/crypto/sha1_generic.c

[3] *"SHA-1 Pseudocode,"*Accessed on: Nov. 8, 2020. [Online]. Available: https://en.wikipedia.org/wiki/SHA-1#Examples_and_pseudocode

[4] *"random(4) - Linux manual page,"*Accessed on: Nov. 10, 2020. [Online]. Available: https://man7.org/linux/man-pages/man4/random.4.html

[5] *"Entropy Management Diagrams,"*Accessed on: Nov. 10, 2020. [Online]. Available: https://eprint.iacr.org/2012/251.pdf

[6] *"A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications,"* Accessed on: Nov. 10, 2020. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf

[7] *"Recoverable Random Numbers in an Internet of Things Operating System,"* Accessed on: Nov. 13, 2020. [Online]. Available: https://www.mdpi.com/1099-4300/19/3/113/htm

[8] *"Ron was wrong, Whit is right,"* Accessed on: Nov. 14, 2020 [Online]. Available: https://eprint.iacr.org/2012/064.pdf

[9] *"Option Pricing - Monte-Carlo Methods,"* Accessed on: Nov. 14, 2020 [Online]. Available: https://www.goddardconsulting.ca/option-pricing-monte-carlo-index.html

[10] "*Random Number Generation Using LFSR*" Accessed on: Nov. 13, 2020 [Online]. Available: https://www.maximintegrated.com/en/design/technical-documents/app-notes/4/4400.html

[11] "*Whitepaper – The Windows 10 Random Number Generation Infrastructure*" Accessed on: Nov. 5, 2020 [Online]. Available: https://www.microsoft.com/security/blog/2019/11/25/going-in-depth-on-the-windows-10-random-number-generation-infrastructure/

[12] "*Special Publication 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators*" Accessed on: Nov. 5, 2020 [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final

[13] "*Reverse-engineered ProcessPrng Code*" Accessed on: Nov. 6, 2020 [Online]. Available: https://github.com/golang/go/issues/33542#issuecomment-564234063

[14] "*SystemPrng function*" Accessed on: Nov. 5, 2020 [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/seccng/systemprng

[15] "*BCryptGenRandom function*" Accessed on: Nov. 5, 2020 [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/bcrypt/nf-bcrypt-bcryptgenrandom

[16] "*CryptGenRandom function*" Accessed on: Nov. 5, 2020 [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptgenrandom

[17] "*RtlGenRandom function*" Accessed on: Nov. 5, 2020 [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/ntsecapi/nf-ntsecapi-rtlgenrandom

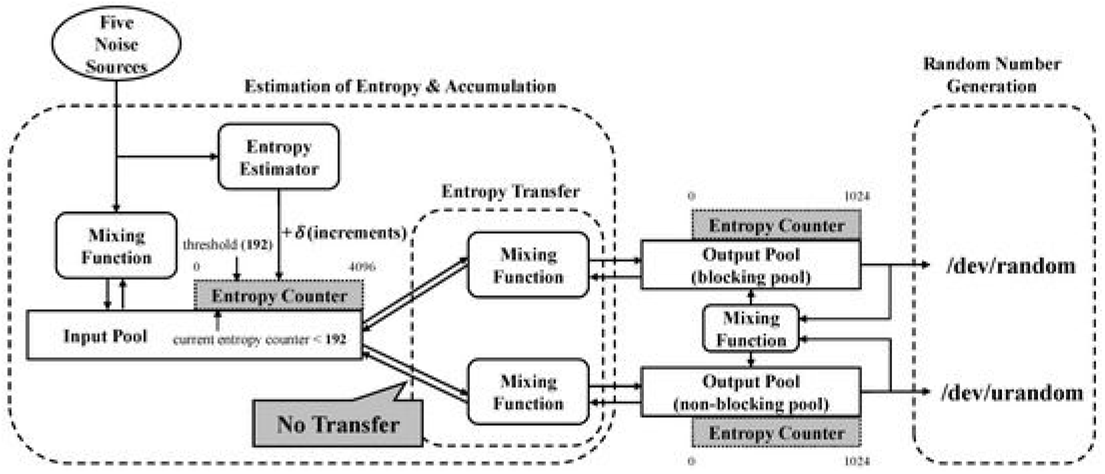
[18] Mohammed Bakiri. *Hardware Implementation of Pseudo Random Number Generator Based on Chaotic Iterations.* Cryptography and Security [cs.CR]. Université Bourgogne Franch-Comté, 2018. English.

[19] *"Intel Digital Random Number Generator (DRNG) Software Implementation Guide"* Accessed on: 11/14/2020 [Online] Available: https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html

9. Appendix A

Figure A.1

Relations between three entropy pools in LRNG.



Source [7]