Windows and Linux Random Number Generation Process: A Comparative Analysis

Khudran Alzhrani
Department of Computer Science
University of Colorado Colorado Springs
Colorado Springs , United States

Amer Aljaedi
Department of Computer Science
University of Colorado Colorado Springs
Colorado Springs , United States

ABSTRACT
In this paper, we explore and analyze the structure and functions of Random Number Generator (RNG) in Windows and Linux opreating systems. And compare the capabilities of their RNGs. It expected that this research would contribute to awareness of the quality and security of the random number generators implemented in Linux and Windows operating systems. It provides unbiased academic research in facilitating informed decision.

General Terms
Random number generator, entropy sources, Security,

Keywords
Windows CNG, True Random Bits, Linux RNG,

1. INTRODUCTION
Random Number generation plays a critical role in cryptography and cryptanalysis. It is essential for many cryptographic tasks such as keys generation, initialization vectors, and cryptographic nonce. Therefore, variety of cryptographic and security applications requires cryptographically strong random numbers, which cannot be predicted or estimated by adversaries. Weak or predictable random number generator can cause catastrophic consequences [1] [2] [3]. Practically, physical resources do not generate sufficient entropy random bits (non-deterministic), in the production environment. Thus, the operating systems uses pseudorandom number generator that seeded by entropy sources. In this order of magnitude, the entropy seed is expanded to generate longer sequences of random numbers that are indistinguishable and statistically unrelated to previous or later generated numbers.

Windows OS Cryptography libraries provide some features that could be used by Windows application developers. Some of those features are encryption, decryption, key storage, hash functions, signature and finally random number generation. Crypt API (CAPI) and its random number generation mechanism, was introduced in Windows XP and earlier versions of Windows. Random number generation in Crypt API used four SHA-1 functions seeded from system entropy. SHA-1's entropy sources and entropy pool reside on Windows Kernel. RC4 is uses by SHA-1 to transmit generated random numbers to the user side and uses MD4 to receive additional entropy provided by the user [4] [5]. Serious vulnerabilities found in Windows XP and Windows 2000 random number generator [6] motivated Microsoft Windows to introduced Cryptography New Generation (CNG) in windows vista. Windows CNG library and its Random number generation are relatively new; there are very few resources and papers regarding CNG random generation mechanism. One paper highlighted some of the CNG library general features and focused on CNG hash functions [7]. Other than that Windows documentation is the sole source that provides a detailed information on Windows CNG RNG.

In Linux environment, Zvi Gutterman et al. [8] published a paper that describes the algorithm of Linux random number generator and identifies vulnerabilities in earlier Linux kernel (version 2.6.10 of the Linux kernel). Due to the lack of detailed documentation for Linux random number generator, the authors stated that they performed combination of static and dynamic reverse engineering to some parts of the Linux kernel source code in order to analyze the algorithm of the generator. Also, they implement a user-mode simulator of Linux RNG to complete their analysis of the generator's behavior. Patric Lacharme et al. [9] discussed the architecture of Linux random number generator with providing mathematical details of the generator properties. In addition, they contacted empirical test of the entropy estimator in Linux random number generator. Yevgeniy Dodis et al. [10] proposed a new security model for pseudorandom number generation. They also provided security assessment of the Linux random numbers generators. Furthermore, they discussed the interfaces of Linux RNG, /dev/random and `/dev/unrandom`. Boaz Barak et al. [11] presented a theoretical model for pseudo-random generation, and they discussed the applicability of their architecture in /dev/(u)random in linux and pseudo-random number generation in smartcards general Requirements for randomness given in RFC 4086 [12]. NIST SP 800-90A [13] provides a detailed guidelines and recommendations for the generation of random bits using deterministic methods.

2. WINDOWS RNG
2.1 CNG Cryptographic Primitive
Most of the applications nowadays utilize cryptographic algorithms either to secure its communications or to provide security services to the end user. Windows has accommodated it Operating Systems with Cryptographic Primitive Libraries such Crypt API in earlier versions of Windows and CNG in the recent ones. Many of CNG algorithms and physical configuration constructed as defined in FIPS 140-2.

CNG functions and interfaces resided at kernel and user modes. CNG.SYS is a cryptographic module that resides in kernel mode and provides cryptographic services to Windows kernel components and kernel mode applications. On the other hand, bcryptprimitives.dll is a cryptographic module that provides cryptographic services to user mode applications. Both CNG.SYS and Bcryptprimitives.dll provide cryptographic services through specific interfaces to offer identical services. CNG introduced many new features such as agility that considered as a threat [7]. Agility in CNG API made it easier for the developers to use and adapt a new algorithm or provider. Cryptographic functions a.k.a. interfaces are used to access CNG API cryptographic services such as a hash function, encryption, decryption,

signature and random number generation. Each cryptographic service is implemented with one or more algorithms. Random Number Generation is treated differently than other cryptographic services because it requires access to Windows kernel in order to retrieve truly random bits generated by Windows entropy sources. Random numbers are used widely in so many well-known cryptographic algorithms such as RC4; also, they are used in SSL/TLS protocols.

In Microsoft Windows 8, kernel mode cryptographic primitive library CNG.SYS shown on Figure 1 is accessed via four logical interfaces CNG BCrypt, Legacy API, SystemPrng interface and Entropy API. Entropy API logical interface is used to collect truly random bits generated from entropy sources to supply the Deterministic Random Bit Generator (DRBG) algorithm reside on CNG.SYS. Random numbers generated by CNG.SYS are utilized as seed to user or kernel mode PRNG through SystemPrng logical interface [14].

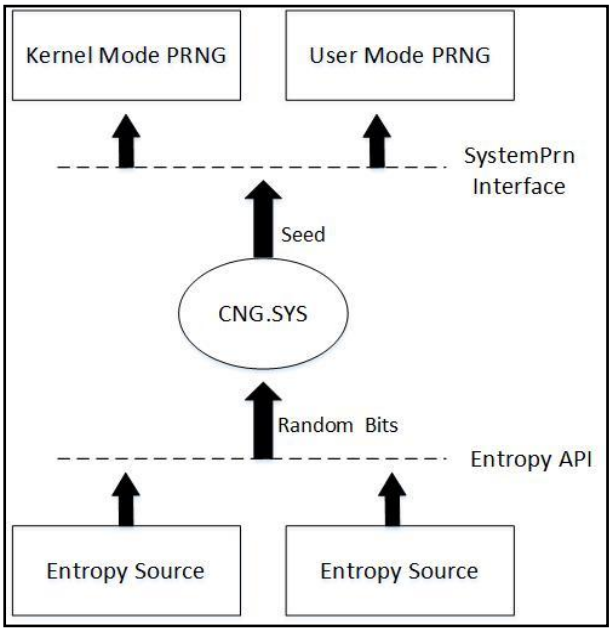

Figure 1: CNG Boundaries

2.2 Windows Entropy Generation

2.2.1 At Booting Time.

As illustrated in Fig 2. Entropy pool is where entropy values are stored and processed. At Windows booting time, an initial entropy value is provided by Windows OS loader. Windows load CNG.SYS and execute the cryptographic operations at the booting time. Windows OS loader collects values from following entropy sources : content of register value HKLM\System\RNG\Seed, random number generated by calling TPM_GetRandom if TPM is available, current system time, OEM0 ACPI table contents, output of RDRAND CPU instruction if supported, output of the UEFI random number generator if booted from UEFI firmware, CPU timing and optionally content of the registry value HKL\System\RNG\ExternalEntropy provided by administrator. Entropy source values are gathered, combined and conditioned by SHA-512. Conditioning the entropy source with SHA-512 distils the entropy into more uniformly and nonbiased samples. According to Windows 8 security policy for FIPS, SHA-512 is non-approved but allowed to be used as NDRNG to seed another approved RNG algorithm. SHA-512 can take up to 2128 as entropy input size and divide entropy input into 1024-bit block pairs in order to produce 512 bits as an output that stored in the entropy pool. [13].

In Windows, there are several kind of tests performed on entropy sources or algorithm such as health test and known answer test. Non-DRNG or SHA-512 and all other RNG algorithms come with a continuous RNG test applied to RNG output. Continuous test performed as following: first blocks of generated n bits compared with the next blocks of n bits. The same comparison procedure is applied to the subsequent n-bit blocks. If any two of n-bit block sequences are equal, then the test result in failure. The Blocks of n-bit that passed continuous RNG test sent to Windows kernel. The kernel will populate entropy pool with true random bits generated by entropy source implemented by Windows OS loader at booting time [15].

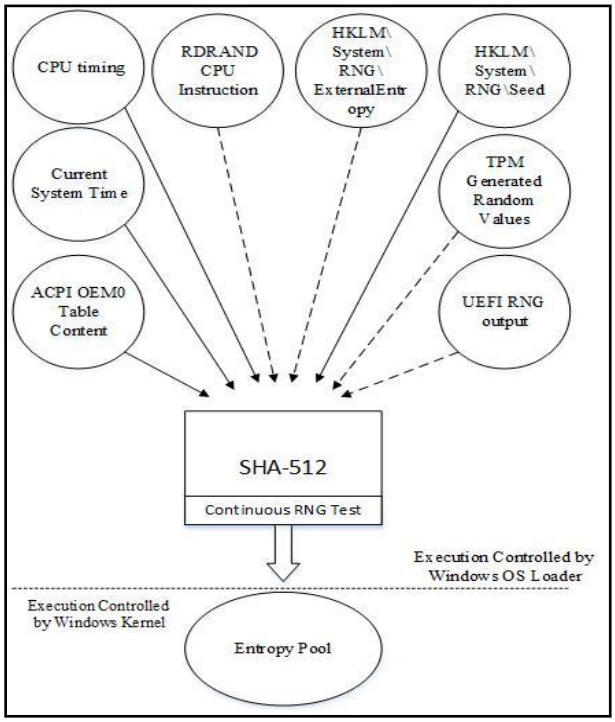

Figure 2: Entropy Generation at Booting Time

2.2.2 At Regular Time.

Meanwhile, kernel and all other necessary files are loaded by Windows OS Loader into the memory, and their integrity are verified.Therefore, the execution control of Windows instructions is granted to Windows kernel and the Windows OS loader terminates its execution [16]. Then, Windows entropy pool is filled with random values from three other different resources high-resolution CPU cycle counter, Trusted Platform Module, and RDRAND CPU instruction. Those values are sent to the entropy pool without using any hash function or conditioner [14].

2.2.3 RdRand Instruction

RdRand is one of the entropy sources deployed by Windows 8 entropy collector. It's quite useful to understand how RdRand instruction is generating random numbers. Intel Processors are widely supported by Windows OS; many of those recently implemented processors support RNG RdRand instruction, which meets FIPS 140-2 and NIST SP800-90A requirement. Generating random numbers with RdRand instruction is quite similar to Windows random number generator process with some differences. According to Intel's digital random number generator implementation guide, all the component

used in RNG are hardware sources. Starting with the entropy source that generates random bits at the rate of 3 GHZ. Then those produced random bits conditioned by AES Block Cipher with CBC mode MAC. 256 bits of conditioned entropy seeds AES CTR_DRBG without the need to use a pool as an intermediate.The reason is that the conditioner can produce high-quality entropy stream with high speed, which is faster than processing entropy that collected in the entropy pool. AES CTR_DRBD generates a random number that seeded by AES-CBC-MAC conditioner [17].

2.2.4 External entropy

Windows Application developer can provide their entropy pool with additional entropy values specified in-kernel mode code. Registered entropy sources could be resided either within windows kernel or any other external device. Each entropy source provider needs to be registered every time the OS is booted because registration of entropy source doesn't hold after rebooting. Registered entropy sources populate entropy pool with random values that in addition to Windows entropy sources values. Windows developer can provide additional entropy values as long as it desired, and then registered entropy sources need to be unregistered[18][19].

2.3 AES CTR_DRBG

Entropy sources value accumulated in Windows entropy pool and periodically requested to be used as a seed for a cascade of two or four AES-256 bits in counter mode based on DRBG. AES-256 CTR is an RBG based on block cipher implemented as defined in NIST special publication 800-90 [14]. CTR_DRBG consists of five functions and an internal state. Update function is called by instantiate and reseed functions and used to update or erase the internal state of CTR_DRBG whenever an initial seed or reseed is requested. Instantiate and reseed functions within CTR_DRBG take entropy values as an input and produce a seed that should be kept as a secret and only used once. While AES-256 CTR_DRBG uses 128 bit as block length and 256 as key length, at least 128+256=384 bits are used as seed length. It is worth mentioning that Windows require any DRBG to be seeded with at least 256 bits of entropy. CTR_DRBG is reseeded periodically or after 248 bit of random number is generated from the current seed. CTR_DRBG fourth function produces and stores random numbers based on the number requested by SystemPrng interface. Also, there is derivation function creates a new seed from a previous seed martial this function is only required in case of no sufficient entropy seed is provided. According to NIST AES-256 CTR_DRBG that it could be implemented without including derivation function. Internal state of AES-256 is used to store some values related to AES algorithm and other functions. Continuous RNG test is performed on the output of AES-256 CTR_DRBG. The Continuous RNG test used here is the same as the one previously explained, with the exception that random number generated is stored in a buffer pointed by SystemPrng Interface[14][13][15]. As shown on Figure 3, Number of AES depends on whether random number is requested from user or kernel mode. In kernel mode cascade of two AES-256 CTR based on DRBG will be seeded and produce random bits, while in user mode a cascade of four AES-256 with counter mode based on DRBG generates random number bits.

2.4 SystemPrng Interface

SystemPrng is a function used to fill a buffer with the random bytes generated by AES CTR_DRBG (see Figure 3). SystemPrng is a Boolean function and it has two parameters namely pbRandomData and cbRandomData. The cbRandomData parameter requests specific number of random bytes to be retrieved from AES-256. The retrieved random numbers are stored in buffer with address pointed at by pbRandomData parameter. SystemPrng function can only be directly called from kernel mode. However, SystemPrng function can also be called indirectly using BCryptGenRandom function. BCryptGenRandom is an exported function to both kernel and user mode that seeds an algorithm provider chosen by the developer with random numbers populated in SystemPrng. Algorithm providers can be initiated and created by BCryptOpenAlgorithmProvider function [14] [18].

2.5 BCryptGenRandom Function

Respectively, CNG.SYS and bcryptprimitives.dll provide their cryptographic operation services by exporting interfaces or functions to kernel and user mode. It is relatively simple to generate random numbers in Windows. The developer has to load and create an algorithm provider using BCryptOpenAlgorithmProvider function. Next, BCryptGenRandom function generates random numbers by the algorithm provider specified in BCryptOpenAlgorithmProvider (See Figure 4). Finally, BCryptCloseAlgorithmProvider function terminates the algorithm provider.

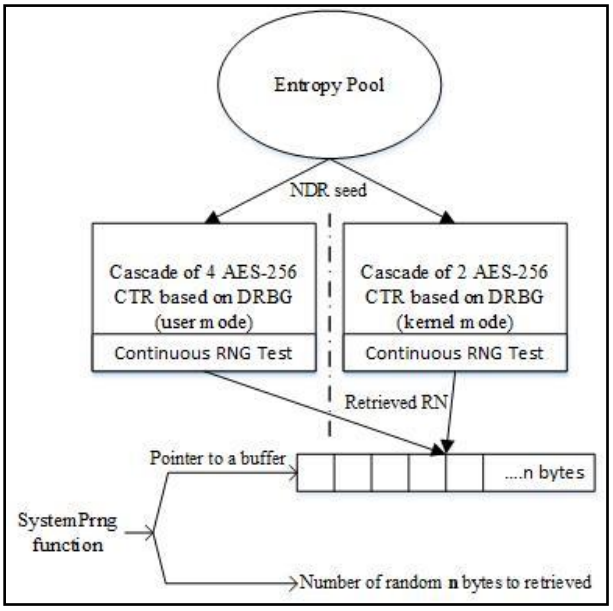

Figure 3: Cryptographic Random Number Generation

Let's take a look at those steps in more details, BCryptOpenAlgorithmProvider function is used for loading CNG providers. In BCryptOpenAlgorithmProvider function takes four parameters. The first parameter is a pointer to a variable where RNG algorithm providers handle is going to be stored. The variable is defined as BCRYPT_ALG_HANDLE. Second parameter is an an identifier for RNG algorithm provider that will generate random bits. There are two different RNG providers in CNG. Each RNG algorithm provider implements a single RNG algorithm. Some providers in CNG can represent more than one cryptographic algorithm, and that is not the case for RNG providers. Other two BCryptOpenAlgorithmProvider function parameters are irrelevant to our topic [14] [18] [18]. All algorithm providers seeded from the buffer pointed at by SystemPrng function and then a continuous RNG test is performed on RNG output.

CNG RNG algorithm providers are as following: Default RNG algorithm provider is AES-256 counter mode and its CNG identifier "BCRYPT_RNG_ALGORITHM". The structure of AES-256 CTR_DRBG is already explained [8]. Dual_EC_DRBG is another algorithm provider that can be identified with BCRYPT_RNG_DUAL_EC_ALGORITHM. Dual_EC_DRBG is based on number theoretic hard problem: given points P and Q on an elliptic curve of order n, find a such that Q = aP. Dual_EC_DRBG can use P-256, P348 or P521 as size of the base field or seedlen with output block length equals to 240, 368 and 504 respectively. Minimum required entropy for instantiate or reseed is 256 bits or equal to maximum bits length of the keys. Dual_EC_DRBG uses similar functions to the one used with AES-256 CTR_DRBG. Two points need to be selected over a filed with at least 2256 in size, In order to start the initiation function or reseed function. Furthermore, generate function also used in Dual_EC-DRBG. In case no seed was provided a derivation function can be applied which employ hash function to create a new seed from previous martial seed [13].

After RNG algorithm provider is loaded, the second step is to generate a random number by calling BCryptGenRandom interface. BCryptGenRandom is a control input interface used by both CNG.SYS and bcryptprimitives.dll, in other words, BCryptGenRandom is exported to both user and kernel modes. BCryptGenRandom function used to fill a buffer specified by OS component or Windows application developer with random number generated by one of the two RNG algorithms introduced previously. Those two RNG algorithms are seeded from buffer pointed by SystemPrng interface. BCryptGenRandom output is the final random number generated which can also be used for key generation. Random number generated by the specified Algorithm provider is also tested using continuous RNG Test. As depicted in Figure 4, BCryptGenRandom function takes four parameters; the first parameter is a handle of algorithm provider which is the BCRYPT_ALG_HANDLE variable defined in BCryptOpenAlgorithmProvider. Second parameter is address of the buffer that receives generated random numbers. Third parameter is number of random number to be generated or the buffer size. The last parameter is flag to change the function behavior. The flag parameter can be one of the following; set to zero, BCRYPT_RNG_USE_ENTROPY_IN_BUFFER which means use the generated numbers as an additional entropy,or BCRYPT_USE_SYSTEM_PREFERRED_RNG to use the system preferred random number generator algorithm[14][18][19].

3. LINUX RNG

3.1 General Structure

The main components of Linux random number generator are three pools which contain the internal state of RNG, and three procedures that control the input and the output to/from the generator and between the pools of the generator. These procedures, also known as functions are described in section III.C, section III.D and section III.E. The general structure of Linux RNG is depicted in figure 5. The three pools have different names based on their functionality. For instance, the first one is entropy pool (or input pool), the second pool is called blocking pool, and the third pool is nonblocking pool. As we can infer from the pools' names, the input pool contains the entropy values that are collected from external entropy sources. The size of this pool is 512 bytes (4094 bits) that contains 128 of 32-bits words. Both the blocking and nonblocking pools are output pools, they reseeded and refreshed from the entropy pool and they have the same size which is 128 bytes and so each contains 32 of 32-bits words. All of the three pools has entropy counter which estimates the amount of entropy in the pool and incremented when entropy value are add and mixed into to the pool. This counter is decremented when random bits are extracted from the pool, the maximum value of this counter is the pool size which indicates that the pool is full.

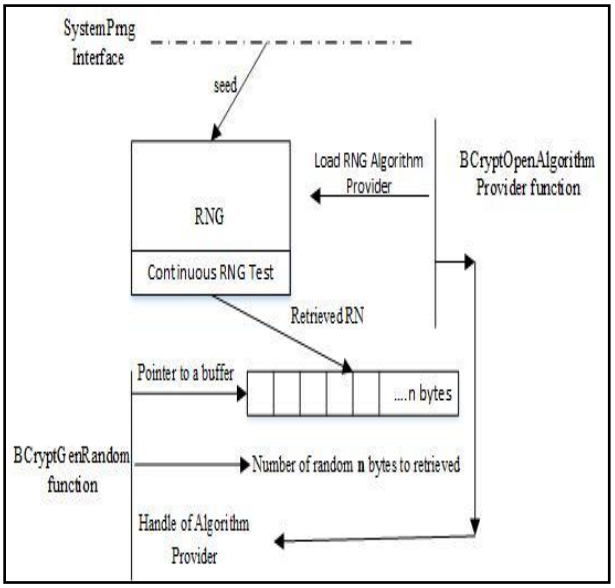

Figure 4: RNG from Developer Point of View

The difference between these two output pools is that the blocking pool will stop generating PRN when there is on enough entropy (based on the entropy counter of the pool) and it waits until more entropy is transferred (reseeded) from input pool, while the nonblocking pool will return as many random bytes as requested without blocking even when there no enough entropy. In case, the nonblocking pool does not receive enough entropy, it will hash the content of its pool using SHA-1, and the digest is mixed back to update the pool (in this case the entropy counter of the nonblocking pool will not increment).

In Linux environment, the API of RNG consists of two character devices, `/dev/random` which is the interface for the blocking pool and `/dev/urandom` the interface for nonblocking pool. Both interfaces are accessible from user space. For kernel space, function `get_random_bytes()` supplies the other Linux components with the requested random bytes. This later function available only for Linux functionalities [8] and it extracts random numbers from nonblocking pool. The `/dev/random` is used to output random bits from blocking pool while the interface `/dev/urandom` outputs bits from nonblocking pool. The designers of the Linux RNG commented about the difference between the two character devices in source code of the random number generator [20] "/dev/random is suitable for use when very high quality randomness is desired (for example, for key generation or one-time pads), as it will only return a maximum of the number of bits of randomness (as estimated by the random number generator) contained in the entropy pool. The `/dev/urandom` device does not have this limit, and will return as many bytes as are requested. As more and more random bytes are requested without giving time for the entropy pool to recharge, this will result in random numbers that are merely cryptographically strong."

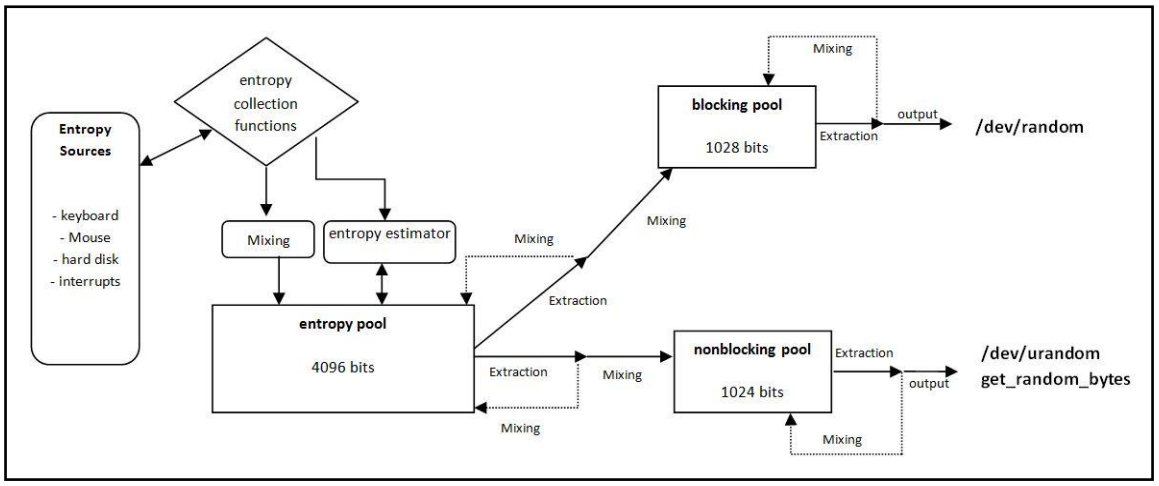

Figure 5: General structure of Linux RNG

3.2 Entropy Collection

The Linux RNG relies on the kernel to gather environmental noise from different devices in the machine as entropy sources. These entropy values are used to initialize and seed the generator [20]. The current implementation of Linux kernel 3.14 uses four functions to gather the entropy values:

add_device_randomness(): this function gathers different values (e.g. MAC addresses, serial numbers and Real Time Clock [2]) that can be used only to initialize the input pool. It used for devises that have little entropy such as Linux on embedded systems.

`add_input_randomness()`: it collects the entropy from input hardware (e.g., keyboard and mouse activities.)

`add_interrupt_randomness()`: this function is called when there is an interrupt, so it uses the value of the interrupt event as entropy input for random generator.

`add_disk_randomness()`: this function collect entropy events from hard disk when the hard derivers call it, it gathers the seek time of block layer request events.

When the entropy event occurs, three 32-bits values are used from that event as entropy input. The num value which depends on the type of the event (e.g. the pressed key or the position of the mouse), The jiffies value which is the time of the event since the system was last booted [9], and the last value is the CPU cycle.

3.2.1 Initialization

When the system starts up, it goes though certain sequence of routines that limit the range of entropy events, and make them predictable, especially the entropy events that originate for hard disk which are deterministic [9]. For this reason, the authors of the Linux generator highlight the importance of using initialization script, which is packaged with most Linux distribution. The script runs as one of the system sequences when the system starts-up or shuts-down. Before the system goes down the script reads and copies 512 bytes from `/dev/urandom` and save it in a file. When the system starts up, it reads from this file and copies the 512 bytes to `/dev/urandom`. The `/dev/urandom` is a rewritable character device and writing into this interface will update the content of the input pool by the same amount of bytes as if they were entropy values [8]. Moreover, these 512 bytes will be added and mixed into the entropy pool by the Mixing function; described in section III.D. Consequently, this effect will also update the blocking and nonblocking pools since they are refreshed and reseeded from the input pool. The added bytes via writing into `/dev/urandom` will not increment the entropy estimator of the input pool (the entropy estimator described in section III.C).

3.3 Entropy Estimation

The estimation of entropy is curial part in generating random numbers, especially for `/dev/random` as we mentioned earlier, this interface will not return random bits if there is not enough entropy in the blocking pool. The entropy counter in blocking and nonblocking pools is incremented by the same amount of bits that transferred from the input pool to each one. When bits are extracted from one of these small pools, the entropy counter of that pool will be decremented by the same amount of the extracted bits.

```
Mixing Algorithm ( pool , i , e ) :
//extent byte to 32-bit word
w = ext(e)
// bitwise left rotation by rotate bits
w = w <<< rotate
w = w xor pool [ i ]
w = w xor pool [ ( i +1) mod 32]
w = w xor pool [ ( i +7) mod 32]
w = w xor pool [ ( i +14) mod 32]
w = w xor pool [ ( i +19) mod 32]
w = w xor pool [ ( i +26) mod 32]
pool [ i ] = ( w >> 3) xor twist_table [w mod 7];

// The  twist_table is definded as follows:

twist_table[8] = {
0x00000000,
0x3b6e20c8,
0x76dc4190,
0x4db26158,
0xedb88320,
0xd6d6a3e8,
0x9b64c2b0,
0xa00ae278}

if i = 0 then
rotate = rotate + 14 (mod 32)
else
rotate = rotate + 7  (mod 32)
```

Figure 6: Pseudo-code of the Mixing function for blocking pool and non-blocking pool. i is index of the current position in the pool, e is the added entropy in byte, twist_table is a table with 8 constant words (each 32 bit). [20]

The entropy estimator of input pool is more complicated and designed in a way that avoids overestimating the amount of collected entropy. Patric Lacharme et al. [9] characterize this estimator as "pessimistic" estimator. As described in the source code [20] and in [9], the entropy estimator of the input pool uses the time delay of the added entropy event (`jiffies`) in three levels and computes the estimation of the added entropy as follows:

Let T_i denote the time of the now event, T_{i-1} the time of the last event.

The first level of the time delay $\delta_a = T_i - T_{i-1}$

The second level $\delta^2_a = \delta_a - \delta_{a-1}$

The third level $\delta^3_a = \delta^2_a - \delta^2_{a-1}$

Then, the estimator will calculate the minimum absolute value of the above deltas $\log_2$:

The entropy added by the event in T_i is

$\log_2 (\min | \delta_a |, | \delta^2_a |, | \delta^3_a |)$.

Therefore, the estimator of the input pool stores the T_{i-1} , δ_{a-1} , δ^2_{a-1} between two events that generated from the same entropy source, so the estimation of the entropy is performed separately for each entropy source. In the end, all of these estimations is summed up, and the entropy estimator incremented accordingly. This estimator is decremented when input pool transfers bits to one of the output pools. According to the source code of the Linux RNG (kernel 3.14) [20], when the input pool became full, RNG will send the new coming entropy to the output pools reciprocally; until the two output pools are 75% full.

3.4 Mixing Function

Whenever entropy added to the input pool or one of the output pools, this entropy value is not added directly to that pool, instead it is mixed and diffused into the pool by the mixing function, which is LFSR-like function. This function is based on modified TGFSR [8] (Twisted Generalized Feedback Shift Register [21][22].) The designers of Linux RNG (kernel 3.14) declared in the source code that they decided to use CRC-32 polynomial in the implementation of twist_table (see figure 6) for maximal mixing and period over $GF(2^{32})$. Also, there are additional polynomials used by mixing function that was chosen base on the size of the pool. Therefore, the input pool polynomial is $(X128 + X104 + X^{76} + X^{51} + X^{25} + X + 1)$, and the blocking and nonblocking pools polynomial is $(X^{32} + X^{26} + X^{19} + X^{14} + X^7 + X + 1)$. The authors of Linux RNG justified their chose of this method for mixing the entropy value [20] as follow "All that we want of mixing operation is that it be a good non-cryptographic hash; i.e. it not produce collisions when fed random data of the sort we expect to see. As long as the pool state differs for different inputs, we have preserved the input entropy and done a good job." The mixing function is used by all of the three pools (e.g. when entropy added to the input pool or entropy transferred to one of the output pools.) Figure 6 shows the algorithm of the mixing function for output pool. First, when a new entropy is received by the input pool, each byte of this entropy value will be extended to 32-bit word, using standard C implicit cast [10]. This 32-bit word will be left rotated by 7 or 14 bits; this based on the current input position in the pool. After that, it will be mixed (using xor) with other seven words in the pool. These seven words are chosen based on the polynomial discussed above. Then, multiplication in $GF(2^{32})$ will be applied by using the lookup table, named "twist_table" in the source code. Afterword, this 32-bit word will be added to the current input position in the pool. It is important to mention that the early versions of Linux RNG use slightly different polynomial, which are $(X^{128} + X^{103} + X^{76} + X^{51} + X^{25} + X + 1)$ for input pool, and $(X^{32} + X^{26} + X^{20} + X^{14} + X^7 + X + 1)$ for output pools. In 2012, Patric Lacharme et al. [9] published a paper with mathematical analysis of the early chosen polynomials for the mixing function. This paper highlights that these previously used polynomials do not achieve the maximal period since they are not primitives over $GF(2^{32})$, and they discussed and provided the right polynomials that generate the maximal period. This paper has taken attention of Linux developers, and they made these polynomials irreducible according to Patric et al. paper in latest Linux RNG (kernel 3.13 and 3.14).

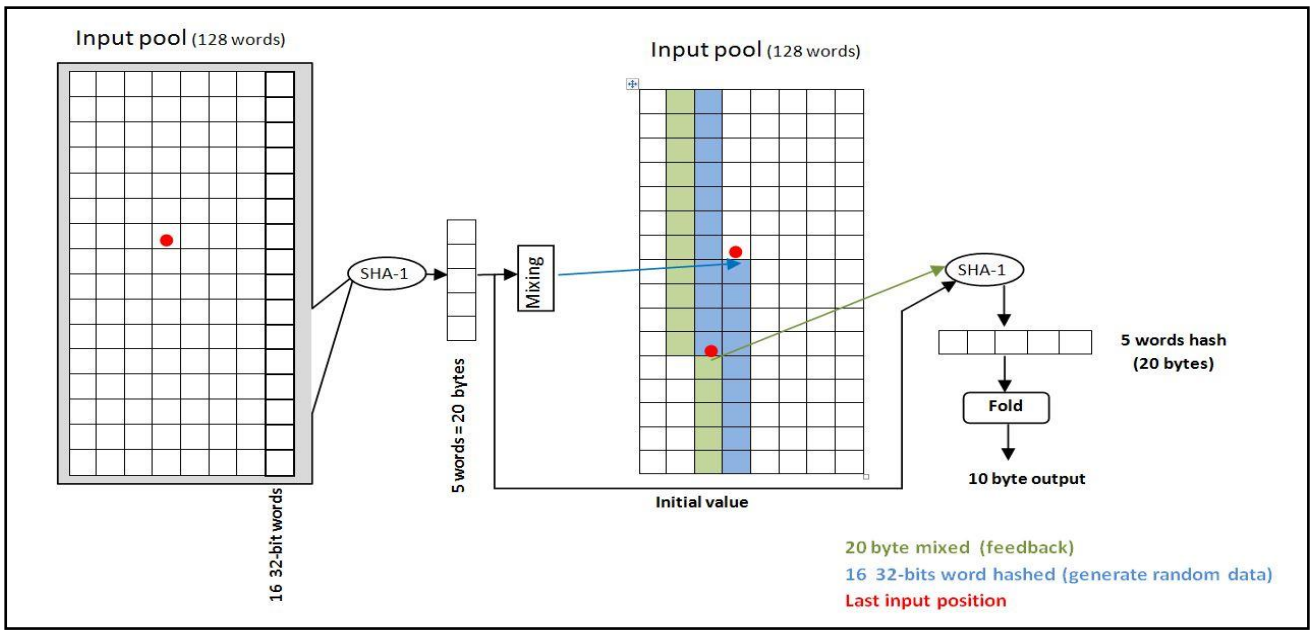

Figure 7: The Extraction function of the Linux RNG.

3.5 Extraction Function

This function is used when random bits are requested via the generator interfaces /dev/random, /dev/urandom ,kernel call get_ramdom_byte, or when the entropy transferred from the input pool into one of the output pools (e.g in case the output pool does not have enough entropy). This output function is considered the only non-linear cryptographic operation used in Linux RNG as it uses SHA-1 to generate the requested random bits and update the internal state of the pool in a feedback manner. It also decrements the entropy counter of that pool by the number of extracted bits. The generation of the random bytes is performed in blocks of 10 output bytes. If the requested random number is not a multiple of 10 bytes, then the last bytes will be truncated according to the number of the requested random bytes. Figure 7 shows the steps of generating random bytes from input pool. As we can see, the SHA-1 is used twice in the extraction function. First, the whole pool is hashed by SHA-1, and the hash digest (5 words) is mixed back to that pool by the mixing function. This feedback (the mixed 20 bytes) updates and shifts 20 words of the pool. This means that for every generated 10 bytes of random number, 20 words in the pool will be updated and shifted (total 640 bits are affected). This is one of the mechanisms used to prevent backtracking (backtracking resistance [9]). In case an adversary knows the internal state of the pool and the current outputs, it will be very difficult for the attacker to guess the previous outputs of that pool as the state of the pool is changed and updated whenever random bits are extracted from it. Also, if the generator receives sufficient entropy inputs, It will be more difficult to predict the future output. In the second step, other 16 words (512 bits) will be extracted from the pool starting from the last input position. The second SHA-1 will hash These words. Also, the hash digest that produced by the first SHA-1 above will be used as initialize value for the second SHA-1. The output digests five words (20 bytes) of the second SHA-1 are folded in half to generate 10 bytes as follow:

W_0, W_1, W_2, W_3, W_4 denote the five words generated from the second SHA-1 [8].

Then, they will be folded by using XOR {W_0 XOR W_3, W_1 XOR W_4, W_2[0-15 bits] XOR W_2[16-31 bits]}

Finally, these 10 bytes will be outputted, and the entropy counter in that pool will be decremented.

4. COMPARATIVE ANALYSIS

4.1 Entropy Sources

Windows 8 utilizes a large number of entropy sources to maintain high entropy in the pool. Large number of those entropy sources doesn't depend on the user input and provides continues entropy including TPM random generator. An experimental research on several TPM chips showed that the entropy was not affected by the number of random bytes generated [23]. Although there are a lot of entropy sources in computer that can be utilized by RNG, the kernel of Linux shortens its entropy collection on limited sources, in compared to windows. Linux RNG receives entropy from four collection functions described in section III.B. Some of them will generate low entropy when Linux machine works as a server (e.g., no user input from keyboard or mouse.) Moreover, in the production environment Linux kernel itself consumes alot of entropy through calls `get_ramdom_bytes` [24]. This highlights the importance of the need to other entropy sources as possible in order to feed the generator with sufficient and continues entropy input.

4.2 Standards and Best Practices

The Linux RNG was designed with reference to some best practice such as RFC 1750, which actually became obsolete by RFC 4086, and the design of the generator does not follow more acceptable standards such as NIST 800-90A. On other hand, Windows 8 is compliant with well-known standards FIPS 140-2 and NIST SP 800-(90A, 90B).

4.3 RNG Performance

The implementations of Mixing and Extraction functions in Linux are aimed to speed the process of generating the random values with maintaining the security and quality of randomness. The designer of linux RNG utilized CRC-32-IEEE 802.3 for the implantation of the mixing function, which helps to update the internal states of the pools with less overhead. This feedback mechanism makes Linux RNG distinguish between other generators who rely intensively on the use of the cryptographic hash functions, which would increase the cost of processing and slow the generator. Whereas, the speed of RNG process in Windows depends on the chosen algorithm to generate the random number and the developer implementation.

4.4 Algorithm Security Strength

One of the most influential aspects that affect the quality of generated random number is the security strength of the deployed random number generation algorithms. There are four different mechanisms in Windows RNG lifecycle that adopted standardized cryptographic algorithms. SHA-512 used for combining the entropy sources values gathered via an entropy source implemented by Windows OS Loader during booting time. AES 256 and Dual EC are applied in different phases of RNG lifecycle in order to generate random number or seed. Cryptographic security strength of those algorithms defined the overall security of Windows RNG. According to NIST Special Publication 800-57 part 1, SHA-512 and AES-256 have 256 bits of security strength which is acceptable until 2031 and beyond. In the case of elliptic curve depends on chosen size of the field or seedlen. For example, 521 bits of seedlen equals to 256 bits of security strength. Linux RNG deploy non-cryptographic algorithm for Mixing function (CRC-32-IEEE-802.3) and SHA-1 for Extraction function. The Mixing function updates the internal states of the pools whenever Extraction function is called and when a new entropy added to the input pool or entropy transferred from the input to output pools. Also, as we discussed earlier, the extraction process uses the SHA-1 twice. The first hash digest is used to feedback the pool and the second hash digest used to generate the random bytes. Therefore, the combination of these two functions together provides backtracking resistance, which makes predicting the internal states of the pools difficult.

4.5 Booting Entropy

Windows provide various types of entropy sources, and it performs both mixing and conditioning on collected entropy values to eliminate the possibility of estimating or predicting the initial values that populate the entropy pool. Linux employs a different strategy (that pointed out by Linux developers) which is the assurance of entropy continuity when the system starts up by the initialization step discussed in section III.B.1. This action can assist to avoid using entropy values of the system starting-up routines that could be predictable.

4.6 RNG Tests and Conditions

Windows Performs a series health test on Windows entropy sources to validate its availability and quality. Also, entropy source values are conditioned to meet the requirements and conditions provided by security standards. Furthermore, all random number generator algorithms apply these continues RNG tests on their output to ensure the quality of randomness for the generated number.

4.7 Evaluation

Linux RNG is part of open source project that allows security researchers to evaluate and analyze its security strength. In our analysis, we were able to locate, in the source code, the different features and structures between latest Linux RNG and earlier versions. While other random number generators that included in proprietary software do not allow this opportunity for their users, which raises some security concerns. Windows doesn't provide enough resources and references that might help to evaluate its RNG. Also there are no research papers on RNG functions in CNG.

4.8 Variety of RNG Algorithms/Interfaces

Developer can choose from two RNG algorithms either AES-256 or Dual EC. This Variety is not provided by most other operating systems nowadays. The Linux RNG provides two output interfaces for user space, which gives the user variety of choices of high-quality randomness over speed and vice versa. If the user prefers high availability over the quality of randomness, then he can use `/dev/urandom` as this interface was designed to provide random output as much as the user request without blocking.

4.9 User Authentication

Windows CNG and Linux don't authenticate the users and processes that mean anyone with legitimate or compromised privileges can implement and use cryptographic services provided by the operating system.

4.10 Security Threats

There are many research papers that targeted RNG process and studied the impact of such weaknesses, but there is another kind of threats that take advantage of the algorithm properties. For example, some of the documents leaked by Edward Snowden stated that NSA had inserted a backdoor into Windows RNG algorithm (Dual_EC_DRBG) which is standardized by NIST. Also, it has been discussed in the literature [4] that denial of service attack could be mounted in blocking pool (in Linux RNG) if the attacker were able to gain access to the system.

4.11 Pool Accessibility

Windows OS restrict its users from accessing or manipulating the entropy pool or even retrieving random bits generated by the internal RNG. While, the interface /dev/urandom in Linux RNG is writable by any user who has access to the system.

4.12 Secure Programming

Windows CNG uses several interfaces that provide an additional layer of security. For instance, Data Input Interface takes data and options as input parameters, but entered data are handled and controlled in the control interface. There are also Output and Status Interfaces.

5. CONCLUSION

Window utilizes cryptographic secure random number generation concept in its RNG process, whereas Linux emphasis on the performance and speed of RNG. Since random numbers are widely used in cryptographic functions, more research need to be done in this area to explore the strength and weaknesses of RNG algorithms and implementation. Moreover, Windows provides its developer with higher entropy availability based on the quality of entropy sources employed in seeding DRBG. This statement also needs to be validated through empirical experiment. Since Linux is an open source and its RNG code is accessible, it is easier to debug and to prove its security strength. Reverse engineering of Windows RNG to test its security is still an open problem that should also considered in future research.

6. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments and suggestions. Also, Special thanks go to Saudi Arabian Cultural Mission in USA for the fund and support.

7. REFERENCES

[1] I. Goldberg and D. Wagner, "Randomness and the Netscape browser," Dr. Dobb's Journal, January 1996. [Online]. Available:http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html.

[2] Z. Gutterman and D. Malkhi, "Hold Your Sessions: An Attack on Java Session-Id Generatio," in A. J. Menezes, (Ed.): CT-RSA 2005, LNCS 3376, pp. 44–57, 2005. [Online]. Available:http://research.microsoft.com/pubs/64680/gm05.pdf.

[3] CVE-2008-0166, "Debian generated SSH-Keys working exploit," [Online]. Available: http://www.securityfocus.com/archive/1/archive/1/492112/100/0/threaded

[4] M. Howard, D. LeBlanc, Writing secure code, Second Ed, Microsoft Press, 2002.

[5] M. Howard, D. Leblanc, and J. Viega. 24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them. McGraw-Hill, New York City, NY, USA, 2009.

[6] L. Dorrendorf, Z. Gutterman, and B. Pinkas. 2007. Cryptanalysis of the windows random number generator. In Proceedings of the 14th ACM conference on Computer and communications security (CCS '07). ACM, New York, NY, USA, 476-485. DOI=10.1145/1315245.1315304.

[7] K. Lee, Y. Lee, J. Park, K. Yim, and I. You, "Security Issues on the CNG Cryptography Library (Cryptography API: Next Generation)," Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2013 Seventh International Conference on , pp.709,713, July 2013.

[8] Z. Gutterman and B. Pinkas, and T Reinman, "Analysis of the Linux random number generator," In IEEE Symposium on Security and Privacy (2006), IEEE Computer Society, pp. 371–385.

[9] P. Lacharme, A. Röck, V. Strubel, and M. Videau, "The Linux pseudorandom number generator revisited," Cryptology ePrint Archive, Report 2012/251, 2012, [Online]. Available: http://eprint.iacr.org/2012/251.pdf

[10] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergnaud, and D. Wichs, "Security Analysis of Pseudo-Random Number Generators with Input: /dev/random is not

Robust," The 2013 ACM SIGSAC conference on Computer & communications security, pp. 647-658.

[11] B. Barak and S. 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.

[12] Randomness Requirements for Security, RFC 4086, June 2005.

[13] E. Barker and J. Kelsey. Recommendation for Random Number Generation Using Deterministic Random Bit Generators, NIST Special Publication 800-90A, January 2012.

[14] Kernel Mode Cryptographic Primitives Library (CNG.SYS), v. 1.1 , Security Policy for FIPS 140-2 Validation, July17, 2013.

[15] Implementation Guidance for FIPS PUB 140-2 and the Cryptographic Module Validation Program. NIST-Communication Security Established Canada. April,2014.

[16] BitLocker® Windows OS Loader (WINLOAD), v.1.1 , Security Policy for FIPS 140-2 Validation , July17, 2013.

[17] Intel Corporation. 2012. Intel Digital Random Number Generator (DRNG) Software Implementation Guide. [Online]. Available: http://software.intel.com/en-us/articles/inteldigital- random-number-generator-drng-softwareimplementation- guide. (Aug. 2012).

[18] Cryptographic Primitives Library (BCRYPTPRIMITIVES.DLL), v.1.1 , Security Policy for FIPS 140-2 Validation , July17,2013.

[19] Microsoft, "Cryptography API: Next Generation", Microsoft Developer Network . [Online]. Available: http://msdn.microsoft.com/enus/

[20] Linux Cross Reference, [Online]. Available: http://lxr.free-electrons.com/source/drivers/char/random.c

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

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

[23] A. Suciu and T. Carean, "Benchmarking the True Random Number Generator of TPM Chips." CoRRabs/1008.2223 (2010) . [Online]. Available: http://arxiv.org/abs/1008.2223.

[24] T. Vuillemin, F. Goichon, C. Lauradoux, and G. Salagnac. " Entropy transfers in the Linux Random Number Generator". hal-00738638, version 1 - 4, 2012.

[25] Linux programmer's Manual. [Online]. Available: http://man7.org/linux/man-pages/man4/random.4.html