# Linux Random Number Generator – A New Approach

Stephan Müller <smueller@chronox.de>

August 12, 2018

**Abstract**

The venerable Linux /dev/random served users of cryptographic mechanisms well for a long time. Its behavior is well understood to deliver entropic data. In the last years, however, the Linux /dev/random showed signs of age where it has challenges to cope with modern computing environments ranging from tiny embedded systems, over new hardware resources such as SSDs, up to massive parallel systems as well as virtualized environments. This paper proposes a new approach to entropy collection in the Linux kernel with the intention of addressing all identified shortcomings of the legacy /dev/random implementation. The new Linux Random Number Generator's design is presented and all its cryptographic aspects are backed with qualitative assessment and complete quantitative testing. The test approaches are explained and the test code is made available to allow researchers to re-perform these tests.

## 1 Introduction

The Linux /dev/random device has a long history which dates all the way back to 1994 considering the copyright indicator in its Linux kernel source code file `drivers/char/random.c`. Since then it provides good random data to cryptographic and even non-cryptographic use cases. The Linux /dev/random implementation was analyzed and tested by numerous researchers, including the author of this paper with the BSI study on /dev/random including a quantiative assessment of its internals [8], the behavior of the legacy /dev/random in virtual environments [10] and presentations on /dev/random such as [9] given at the ICMC 2015. All the studies show that the random data out of /dev/random are highly entropic and thus have a good quality.

So, why do we need to consider a replacement for this venerable Linux /dev/random implementation?

### 1.1 Linux /dev/random Status Quo

In recent years, the computing environments that use Linux have changed significantly compared to the times at the origin of the Linux /dev/random. By using the timing of block device events, timing of human interface device (HID) events

as well as timing of interrupt events[1], the Linux /dev/random implementation derives its entropy.

The block device noise source provides entropy by concatenating:

- the block device identifier which is static for the lifetime of the system and thus provides little or no entropy,
- the event time of a block device I/O operation in Jiffies which is a coarse timer and provides very limited amount of entropy, and
- the event time of a block device I/O operation using a high-resolution timer which provides almost all measured entropy for this noise source.

The HID noise source collects entropy by concatenating:

- the HID identifier such as a key or the movement directions of a mouse which provide a hard to quantify amount of entropy,
- the event time of an HID operation in Jiffies which again provides a very limited amount of entropy, and
- the event time of an HID operation using a high-resolution timer that again provides almost all measured entropy for this noise source.

The interrupt noise source obtains entropy by using:

- mixing the high-resolution time stamp, the Jiffies time stamp, the value of the instruction pointer and the register content into a per-CPU `fast_pool` where the high-resolution time stamp again provides the majority of entropy – due to a high correlation between the interrupt occurrence and the HID / block device noise sources the content of the `fast_pool` at the time of injection into the `input_pool` is heuristically assumed to have one bit of entropy, and
- injecting the content of the `fast_pool` into the `input_pool` entropy pool once a second or after 64 interrupts have been processed by that per-CPU `fast_pool` – whatever comes later.

The interrupt noise source therefore provides very little heuristically estimated entropy considering the number of processed events compared to the other noise sources.

What are the challenges for those aforementioned three noise sources?[2]

The entropy for block devices is believed to be derived from the physical phenomenon of turbulence while the spinning disk operates and the resulting uncertainty of the exact access time. In addition, when accessing a sector on the disk, the read head must be re-positioned and the hard disk must wait until the sector to be accessed is below the read head. Again, the uncertainty of exact

[1] The additional sources of entropy from user space via an IOCTL on /dev/random as well as specialized hardware implementing a random number generator should be left out of scope as they are entropy sources that are not modeled by the Linux /dev/random. Further, as these sources of entropy are rarely available, /dev/random cannot rely on their presence.

[2] Note, the legacy /dev/random implementation also uses information from device drivers via `add_device_randomness`. That function can be considered as a noise source itself. As this data is credited with zero bits of entropy, it is not subject to discussion here.

access time is the root cause of entropy. Let us assume that these assumptions are all correct. The issue in modern computing environments is that fewer hard disks with spinning platters are used. Solid State Disks (SSD) are more and more in use where all of these assumptions are simply not applicable as these disks are not subject to turbulence, read head positioning or waiting for the spin angle when accessing a sector. Furthermore, hard disks with spinning platters more commonly have large caches where accessed sectors served out of that cache are again not subject to the root causes of entropy. In addition, the more and more ubiquitous use of Linux as guest operating system in virtual environments again do not allow assuming that the mentioned physical phenomena are present. Virtual Machine Monitors (VMM) may use large buffer caches[3]. Also, a VMM may convert a block device I/O access into a resource access that has no relationship with hard disks and spinning platters, such as a network request. The same applies to Device Mapper setups. When a current Linux kernel detects that it has no hard disks with spinning platters – which includes SSDs or VMM-provided disks – or Device Mapper targets are in use, the Linux kernel simply deactivates these block devices for entropy collection[4]. Thus, for example, on a system with an SSD, no entropy is collected when accessing that disk.

HIDs are commonly a great source of entropy as they deliver much entropy. Each movement of the mouse by one tick triggers the entropy collection. Also, each key press and release individually generates an event that is used for entropy. However, a large number of systems run headless, such as almost all servers either on bare metal or within a virtual machine. Thus, entropy from HIDs is simply not present on those systems. Now, having a headless server with an SSD, for example, implies that two of the three noise sources are unavailable. Such systems are left with the interrupt noise source whose entropy contribution is rated very low by the legacy /dev/random entropy estimator compared to the two unavailable noise sources. During testing, the author also observed that HIDs attached to the system via USB are not used as entropy source.

As already mentioned, the interrupt noise source's entropy contribution is rated very low compared to the other two noise sources. In addition, a key aspect not to be overlooked is the following: a HID or block device event providing entropy to the respective individual noise sources processing generates an interrupt. These interrupts are also processed by the interrupt noise source. As mentioned above, the majority of entropy is delivered by the high-resolution time stamp of the occurrence of such an event. Now, that event is processed twice: once by the HID or block device noise source and once by the interrupt noise source. Thus, initially the two time stamps of the one event (HID noise source and interrupt noise source, or block device noise source and interrupt noise source) used as a basis for entropy are highly correlated. Correlation or even a possible reuse of the same random value diminishes entropy significantly. The use of a per-CPU `fast_pool` with an LFSR and the injection of the `fast_pool` into the core entropy pool of the `input_pool` after the receipt of

[3] In case of KVM, the host Linux kernel uses its buffer cache which can occupy the entire non-allocated RAM of the hardware.

[4] An interested reader may trace the Linux kernel source code where the flag `QUEUE_FLAG_ADD_RANDOM` is cleared. One of the key locations is the function `sd_read_block_characteristics` that disables SSDs as entropy source.

64 interrupts can be assumed to change the distribution of the input value such that the correlation would be difficult to exploit in practice. Furthermore, the assumption that at the time of injecting of a `fast_pool` into the `input_pool` the contents of that `fast_pool` has only one bit of entropy counters correlation effects. As of now, however, the author is unaware of any quantitative study analyzing whether the correlation is really broken and the `fast_pool` can be assumed to have one bit of entropy.

The discussion shows that the noise sources of block devices and HIDs are a derivative of the interrupt noise source. All events used as entropy source recorded by the block device and HID noise source are delivered to the Linux kernel via interrupts.

## 1.2 A New Approach

Given that for all three noise sources challenges are identified in modern computing environments, a new approach for collecting and processing entropy is proposed.

To not confuse the reader, the following terminology is used:

- The Linux /dev/random implementation in `drivers/char/random.c` is called legacy /dev/random henceforth.
- The newly proposed approach for entropy collection is called Linux Random Number Generator (LRNG) throughout this document.

The new approach implements the modeling of a slow noise source within the LRNG based on the timing of interrupts and allowing other, even fast operating noise sources to contribute entropy. As discussed above for the legacy /dev/random, only the high-resolution time stamps deliver entropy for hardware events and other information processed by the legacy /dev/random implementation hardly have any entropy. This identification is used as a basis that solely the timing of interrupts is used.

The cryptographic processing of the entropic data is implemented with a well-known and well-studied deterministic random number generator: an SP800-90A DRBG as defined in [1] – naturally excluding the NSA-sponsored Dual-EC DRBG. Other DRNGs such as a ChaCha20 based DRNG are possible, too. The concept of the LRNG allows the selection of the used DRNG and the backend cipher implementations at compile time. The use of the SP800-90A DRBG allows the use of assembler or hardware accelerator supported cipher primitives by relying on the kernel crypto API for all cipher primitives including the DRBG implementation itself.

By using the kernel crypto API to provide the cipher primitives, other benefits are reaped, such as the the self-test logic implemented by the kernel crypto API test manager.

The LRNG reaches a cryptographically acceptable seed level much earlier than the legacy /dev/random implementation. Commonly, the minimal entropy threshold of 128 bits of the LRNG is reached before or at the time user space boots. The full seed level of 256 bits is reached at the time the initramfs is executed but before the root partition is mounted on standard Linux distributions.

The key idea is to focus on the collection of entropy from interrupts. Why are interrupts considered? The following answers apply:

- As seen with the discussion of the legacy /dev/random above, the HID and block devices noise sources must be considered as a derivative[5] of an interrupt noise source. Said differently, when focusing on interrupts, the HID and block devices are automatically covered with the exception of the HID and block device specific information like the key ID which contribute as mentioned very little entropy.

- The idea for the LRNG design came the author during a study currently conducted for the German BSI analyzing the behavior of entropy and the operation of entropy collection in virtual environments. As mentioned above, modeling noise sources for block devices and HIDs is not helpful for virtual environments. However, any kind of interaction with virtualized or real hardware requires a VMM to still issue interrupts. These interrupts are issued at the time the event is relayed to the guest. As on bare metal, interrupts are issued based on either a trigger point generated by the virtual machine or by external entities wanting to interact with the guest. Irrespective whether the VMM translates a particular device type into another device type (e.g. a block device into a network request), the timing of the interrupts triggered by these requests is hardly affected by the VMM operation. Thus entropy collection based on the time stamping of interrupts is hardly affected by a VMM.

- By focusing on interrupt timing, only one value that can be obtained without using any other Linux services needs to be processed. That allows implementing a fast code path to process such information. As the recording logic of the LRNG for interrupts must be hooked into a hot code path of the Linux kernel, keeping only a tightly limited amount of LRNG code in that code path benefits the overall performance.

- Considering the other values recorded by the legacy /dev/random implementation contribute hardly any entropy based on the quantitative assessment given with [8], maintaining them requires extra overhead without significant benefit. Hence, they are disregarded in the LRNG design.

Before discussing the design of the LRNG, the goals of the LRNG design are enumerated:

1. During boot time, the LRNG must already provide random numbers with sufficiently high entropy. It is common that long-running daemons with cryptographic support seed their deterministic random number generators (DRNG) when they start during boot time. The re-seeding of those DRNGs may be very much later, if at all. Daemons that link with OpenSSL, for example, use a DRNG that is *not* automatically re-seeded by OpenSSL. If the author of such daemons is not careful, the OpenSSL DRNG is seeded once during boot time of the system and never thereafter. Hence seeding such DRNGs with random numbers having high entropy is very important.

   As documented in section 3.3 the DRBG is seeded with full security strength of 256 bits during the first steps of the initramfs time after about

[5]The term derivative indicates that they are strongly related and does not indicate that the one is a mathematical derivative of the other.

1.3 seconds after boot. That measurement was taken within a virtual machine with very few devices attached where the legacy /dev/random implementation initializes the `nonblocking_pool` after 30 seconds or more since boot with 128 bits of entropy. In addition, the LRNG maintains the information by when the DRBG is "minimally" seeded with 128[6] bits of entropy. This is commonly achieved even before user space is initiated.

2. The LRNG must be a drop-in replacement for the legacy /dev/random in respect to the ABI and API of its external interfaces. This allows keeping any frictions during replacement to a minimum. The interfaces to be kept ABI and API compatible cover all in-kernel interfaces as well as the user space interfaces. No user space or kernel space user of the LRNG is required to be changed at all.

3. The LRNG must be very lightweight in hot code paths. As described in the design in chapter 2, the LRNG is hooked into the interrupt handler and therefore should finish the code path in interrupt context very fast.

4. The LRNG must not use locking in hot code paths to limit the impact on massively parallel systems.

5. The LRNG must handle modern computing environments without a degradation of entropy. The LRNG therefore must work in virtualized environments, with SSDs, on systems without HIDs or block devices and so forth.

6. The LRNG must provide a design that allows quantitative testing of the entropy behavior.

7. The LRNG must use testable and widely accepted cryptography for whitening.

8. The LRNG must allow the use of cipher implementations backed by architecture specific optimized assembler code or even hardware accelerators. This provides the potential for lowering the CPU costs when generating random numbers – less power is required for the operation and battery time is conserved.

9. The LRNG must separate the cryptographic processing from the noise source maintenance to allow a replacement of these components.

## 1.3 Document Structure

This paper covers the following topics in the subsequent chapters:

- The design of the LRNG is documented in chapter 2. The design discussion references to the actual implementation whose source code is publicly available.

- The testing of the LRNG is covered in chapter 3. The testing is performed for various aspects of the LRNG, including a comparison with the legacy /dev/random behavior, the entropy assessment of the raw noise as well as a study of the correctness of the output of the LRNG.

[6] The background for this value is discussed in section 2.5.

- Different ideas on how to integrate the LRNG with the current Linux kernel code tree are discussed in section 4.
- The various appendices cover miscellaneous topics supporting the general description.

# 2 LRNG Design

The LRNG can be characterized with figure 2.1 which provides a big picture of the LRNG processing and components.

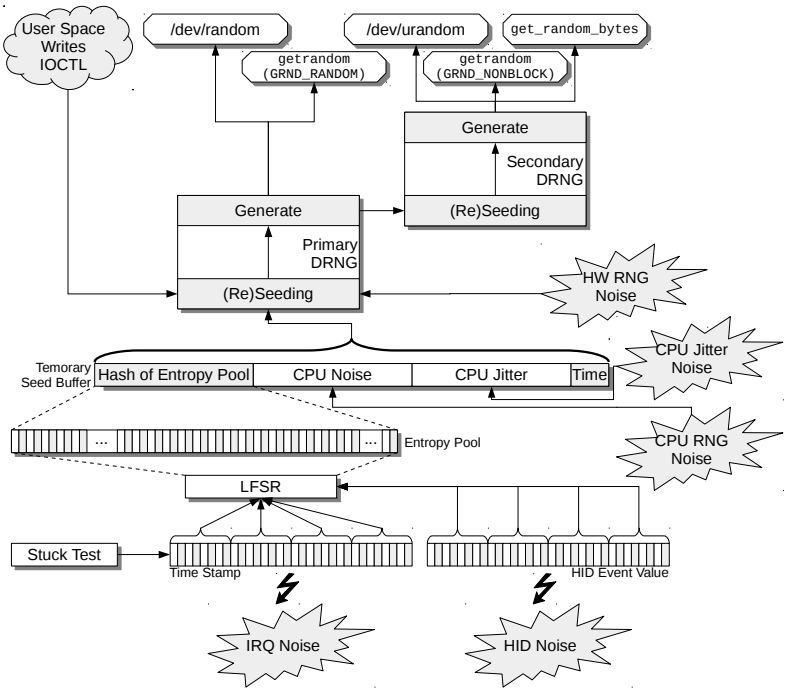


Figure 2.1: LRNG Big Picture

The heart of the LRNG is the maintenance of a deterministic random number generator (DRNG) which are defined at compile time. Currently, an SP800-90A DRBG defined by [1] using the kernel crypto API for its operation is available. In case the kernel crypto API shall not be compiled, the LRNG offers a ChaCha20-based DRNG and SHA-1 for the hash operation which both are implemented in C without any additional dependency. Additional DRNGs may be implemented as the LRNG has a small internal API that needs to be covered. Note, throughout this document, all statements regarding the DRNG equally apply to the SP800-90A DRBG as well as the ChaCha20 DRNG or other yet-to-be-developed DRNGs unless otherwise noted.

Entropy derived from interrupts is injected into the DRNG. The interrupt noise source is the only noise source completely modeled and implemented by the LRNG. Other noise sources that are developed independently from the LRNG

can feed entropic (or even non-entropic) data into the DRNG. Such "LRNG-external" noise sources include:

- If available, the LRNG uses random number generators present in CPUs, such as the Intel RDRAND instruction as a noise source. Note, this noise source is not auditable. Therefore, the LRNG assumes only a small amount of entropy is present with this noise source as documented in section 2.5.1.
- If available, the CPU Jitter random number generator is used as a noise source. The entropy content of the noise source is discussed in section 2.5.2.
- The Linux kernel implements drivers for specific hardware RNGs. The hardware RNG driver framework is able to inject data into the DRNG.
- User space can write data into /dev/random or /dev/urandom which are injected into the primary and secondary DRNG and assumed to have no entropy.
- User space can use the IOCTL of `RNDADDENTROPY` to write data into the primary DRNG with is associated with an entropy value.

Other noise sources can be added with ease – the structure allows and even supports the addition of new noise sources, be it slow or fast noise sources.

The DRNG allows two methods of obtaining random data:

- For users requiring random numbers from a seeded and frequently reseeded secondary DRNG, such as the /dev/urandom, the `getrandom` system call or the in-kernel `get_random_bytes` function, the secondary DRNG is accessed directly by invoking its generate function. This generate function complies with the generate function discussed in SP800-90A.
- Users requiring random data that contains information theoretical entropy, such as for seeding other DRNGs also use the primary DRNG's generate function via the /dev/random device file and the `getrandom` system call when invoked with `GRND_RANDOM`. The difference to the /dev/urandom handling is that:
  1. each primary DRNG generate request is limited to the amount of entropy the of the DRNG was seeded with, and
  2. each DRNG generate request is preceded by a reseeding of the DRNG.

The processing of entropic data from the noise source before injecting them into the primary DRNG is performed with the following mathematical operations:

1. LFSR: The 32 least significant bits of the time stamp data received from the interrupts[7] are processed with an LFSR. That LFSR is identical to the LSFR used in the legacy /dev/random implementation except that it is capable of processing an entire word and that a different polynomial is used. Also, this LFSR is used in the OpenBSD /dev/random equivalent.

[7]Note, also the HID event numbers like pressed key numbers or mouse movement coordinates are also mixed into the entropy pool using this LSFR. As that data is credited with zero entropy, it is not further discussed.

2. Concatenation: The temporary seed buffer used to seed the primary DRNG is a concatenation of parts of the entropy pool data, and the CPU noise source output.

The following subsections cover the different components of the LRNG from the bottom to the top.

## 2.1 LRNG Big Picture

Before going into the details of the LRNG processing, the concept underlying the LRNG shown in figure 2.1 is provided here.

The entropy derived from the slow noise source is collected and accumulated in the entropy pool.

At the time the primary DRNG is seeded, the entire entropy pool is hashed with a hash defined at compile time[8]. The caller is returned the hash truncated to the amount of entropy that is to be given to the caller. In case the hash type generates an output smaller than the entropy amount that can and shall be delivered to the caller, the following steps are used to extract the necessary data:

1. Generate the hash and place it in the output buffer.

2. Mix the hash result back into the entropy pool.

3. Go to step 1 and re-perform the operation until sufficient data is generated.

The entire hash is mixed back into the entropy pool for backtracking resistance. The hash output is concatenated with data from the fast noise sources controlled by the LRNG: the CPU noise source if it is available.

The primary DRNG always tries to seed itself with 256 bits[9] of data, except during boot. If the noise sources cannot deliver that amount, the available entropy is used and the primary DRNG keeps track on how much entropy it was seeded with. During boot, the primary DRNG is seeded as follows:

1. At the time of initialization of the LRNG, the available entropy in the entropy pool and potentially with the fast noise sources are injected into the primary DRNG.

2. The DRNG is reseeded from the entropy pool and potentially the fast noise sources if the entropy pool has collected at least 32 data bits from the interrupt noise source, i.e. one word, the smallest data unit that can be read from the entropy pool. The goal of this step is to ensure that

[8]The examples in the code use a hash that is the same as used for the DRNG type which limits the required cipher support to only once cipher. In case of the CTR DRBG, the CMAC-AES is suggested. The key for the CMAC-AES is set during initialization time by reading a key equal to the AES type used for the DRBG from the content of the entropy pool. That key is not changed afterwards. The idea is that the CMAC AES shall operate like a hash, i.e. compressing and whitening the entropy pool. The behavior of an authenticating MAC is irrelevant for the purpose here which implies that the key does not need to be changed at runtime.

[9]That value depends on the security strength of the chosen DRNG. If the DRNG is the CTR DRBG with AES 128 with a security strength of 128 bits, this value is 128 bits. Note, in the remainder of this document, a DRNG with a security strength of 256 bits is assumed when the value of 256 bits is referred to.

the primary and secondary DRNG receive some initial entropy as early as possible.

3. The DRNG is reseeded from the entropy pool and potentially the fast noise sources if the entropy pool has collected at least 128 bits of entropy.

4. The DRNG is reseeded from the entropy pool and potentially the fast noise sources if the entropy pool has collected at least 256 bits[10] of entropy.

At the time of the reseeding steps, all available entropy from all noise sources is used. This may imply that one or more of the aforementioned steps are skipped. For example, at the time of step 1 and the presence of the CPU noise source, more than 128 bits of entropy are obtained even if no interrupts are collected. This means that the LRNG arrives immediately at step 3, skipping steps 1 and 2.

In all listed steps, the secondary DRNG is (re)seeded with a number of random bytes from the primary DRNG that is equal to the amount of entropy the primary DRNG was seeded with. This means that when the primary DRNG is seeded with 128 or 256 bits of entropy, the secondary DRNG is seeded with that amount of entropy as well[11].

Before the primary DRNG is seeded with 256 bits of entropy in step 4, requests of random data from /dev/random are not processed.

At runtime, the primary DRNG delivers only random bytes equal to the entropy amount it was seeded with. E.g. if the primary DRNG was seeded with 128 bits of entropy, it will return only 128 bits of random data. Subsequent requests for random data are only fulfilled after a reseeding operation of the primary DRNG.

The secondary DRNG operates as deterministic random number generator with the following properties:

- The maximum number of random bytes that can be generated with one DRNG generate operation is limited to 4096 bytes.

- The secondary DRNG is reseeded

  - If the last reseeding of the secondary DRNG is more than 600 seconds ago[12], or
  - $2^{20}$ DRNG generate operations are performed, whatever comes first, or
  - the secondary DRNG is forced to reseed before the next generation of random numbers if data has been injected into the LRNG by writing data into /dev/random or /dev/urandom.

[10] As mentioned, the fully seeded value is equal to the DRNG security strength. That means, this value is set to 128 bits if the CTR DRBG with AES 128 is used.

[11] There is only one exception to that rule: during initialization before the seed level of 128 bits is reached, a random number with 256 bit is generated by the primary DRNG to seed the secondary DRNG.

[12] Note, this value will not empty the entropy pool even on a completely quiet system. Testing of the LRNG was performed on a KVM without fast noise sources and with a minimal user space, where only the SSH daemon was running, During the testing, no operation was performed by a user. Yet, the system collected more than 256 bits of entropy from the interrupt noise source within that time frame, satisfying the secondary DRNG reseed requirement.

The chosen values prevent high-volume requests from user space to cause frequent reseeding operations which drag down the performance of the DRNG[13][14].

When the secondary DRNG requests a reseeding from the primary DRNG and the primary DRNG pulls from the entropy pool, an emergency entropy level of 512 bits of entropy is left in the entropy pool. This emergency entropy is provided to serve /dev/random even while /dev/urandom is stressed.

With the automatic reseeding after 600 seconds, the LRNG is triggered to reseed itself before the first request after a suspend that put the hardware to sleep for longer than 600 seconds.

### 2.1.1 Minimally Versus Fully Seeded Level

The LRNG's primary and secondary DRNGs are reseeded when the first 128 bits / 256 bits of entropy are received as indicated above. The 128 bits level implies that the secondary DRNG is considered "minimally" seeded whereas reaching the 256 bits level implies that the secondary DRNG is "fully" seeded.

Both seed levels have the following implications:

- Upon reaching the minimally seeded level, the kernel-space callers waiting for a seeded DRNG via the API calls of either `wait_for_random_bytes` or `add_random_ready_callback` are woken up. This implies that the minimally seeded level is considered to be sufficient for in-kernel consumers.

- When reaching the fully seeded level, the user-space callers waiting for a fully seeded DRNG via the `getrandom` system call are woken up. This means that the fully seeded level is considered to be sufficient for user-space consumers.

### 2.1.2 NUMA Systems

To prevent bottlenecks in large systems, the secondary DRNG will be instantiated once for each NUMA node. The instantiation of the secondary DRNGs happen all at the same time when the LRNG is initialized.

The question now arises how are the different secondary DRNGs seeded without re-using entropy or relying on random numbers from a yet insufficiently seeded LRNG. The LRNG seeds the secondary DRNGs sequentially starting with the one for NUMA node zero – the secondary DRNG for NUMA node zero is seeded with the approach of 32/128/256 bits of entropy stepping discussed above. Once the secondary DRNG for NUMA node 0 is seeded with 256 bits of entropy, the LRNG will seed the secondary DRNG of node one when having

[13] Considering that the maximum request size is 4096 bytes defined by `LRNG_DRNG_MAX_REQSIZE` (i.e. each request is segmented into 4096 byte chunks) and at most $2^{20}$ requests defined by `LRNG_DRNG_RESEED_THRESH` can be made before a forced reseed takes place, at most $4096 \cdot 2^{20} = 4,294,967,296$ bytes can be obtained from the secondary DRNG without a reseed operation.

[14] After boot, the secondary ChaCha20 DRNG state is also used for the atomic DRNG state. Although both DRNGs are controlled by separate and isolated objects, the DRNG state is identical. As the `LRNG_DRNG_RESEED_THRESH` is enforced local to each DRNG object, the theoretical maximum number of random bytes the ChaCha20 DRNG state could generate before a forced reseed is twice the amount listed before – once for the secondary DRNG object and once for the atomic DRNG object.

again 256 bits of entropy available. This is followed by seeding the secondary DRNG of node two after having again collected 256 bits of entropy, and so on.

When producing random numbers, the LRNG tries to obtain the random numbers from the NUMA node-local secondary DRNG. If that secondary DRNG is not yet seeded, it falls back to using the secondary DRNG for node zero.

Note, to prevent draining the entropy pool on quiet systems, the time-based reseed trigger, which is 600 seconds per default, will be increased by 100 seconds for each activated NUMA node beyond node zero. Still, the administrator is able to change the default value at runtime.

### 2.1.3 Flexible Design

Albeit the preceding sections look like the DRNG and the management logic are highly interrelated, the LRNG code allows an easy replacement of the DRNG with another deterministic random number generator. This flexible design allowed the implementation of the ChaCha20 DRNG if the SP800-90A DRBG using the kernel crypto API is not desired.

To implement another DRNG, all functions in `struct lrng_crypto_cb` in "lrng.h" must be implemented. These functions cover the allocation/deallocation of the DRNG and the entropy pool read hash as well as their usage.

The implementations can be changed at runtime. The default implementation is the ChaCha20 DRNG using a software-implementation of the used ChaCha20 stream cipher and the SHA-1 hash for accessing the entropy pool.

### 2.1.4 Covered Design Concerns of Legacy /dev/random[15]

The seeding approach of the LRNG covers one theoretical problem the legacy /dev/random implementation faces: during initialization time, noise from the noise sources is injected directly into the `nonblocking_pool`. Depending on the assumed entropy in the data, zero to 11 bits of entropy may be stirred into the `nonblocking_pool` per injection. At the same time the `nonblocking_pool` is seeded, callers may read random data from it. For an insufficiently seeded random number generator, this leads to a loss in entropy that is visualized with the following worst case analogy: when an RNG receives one bit of entropy which is followed by a generation of one or more random numbers, the caller requires an attack strength of one bit to break the state of the RNG. When one new bit of entropy is received after the attacker's gathering of random data, the new state of the RNG will again only have one bit of entropy and not two bits (the addition of the first and second seed). Hence, in a pathological case, the `nonblocking_pool` may receive 128 bits of entropy in 128 separate seeding steps where an attacker can request random data from the `nonblocking_pool` between each seeding operation. The attack strength required to break the RNG in this case is $2^1 \cdot 128$ and not $2^{128}$ – i.e. the attack strength is reduced to a manageable level. In case the attack is applied, the `nonblocking_pool` will *not* have 128 bits of entropy, but *zero* bits! The LRNG does not face this problem during initialization, because the entropy in the seed is injected with one atomic operation into the primary and secondary DRNG. The issue alleviated to some extent as during initialization four chunks of 64 bits each

[15] This issue has been addressed to some extent by sending four 64 byte segments from the `fast_pool`s to the ChaCha20 DRNG at boot time with the kernel version 4.8.

derived from the interrupt noise source are injected into the ChaCha20 DRNG starting with Linux kernel 4.8.

With the legacy /dev/random implementation in case /dev/urandom or `get_random_bytes` is heavily read, a user space entropy provider waiting with `select(2)` or `poll(2)` on /dev/random will not be woken up to provide more entropy. This scenario is fixed with the LRNG where the user space entropy provider is woken up.

## 2.2 LRNG Data Structures

The LRNG uses three main data structures:

- The interrupt noise source is processed with an entropy pool. That entropy pool has various status indicators supporting the interrupt processing to obtain entropy. To ensure that no locking is needed when accessing this entropy pool in hot code paths, all relevant data units are `atomic_t` variables. This includes the entropy pool itself which is an array of `atomic_t` variables that are all processed with the available atomic operations. By using atomic operations, locking is irrelevant, especially in the hot code paths.

- The deterministic random number generator data structure for the primary DRNG holds the reference to the kernel crypto API data structure DRNG and associated meta data needed for its operation.

- The secondary DRNG is managed with a separate data structure.

## 2.3 Interrupt Processing

The LRNG hooks a callback into the bottom half interrupt handler at the same location where the legacy /dev/random places its callback hook.

The LRNG interrupt processing callback is a void function that also does not receive any input from the interrupt handler. That interrupt processing callback is the hot code path in the LRNG and special care is taken that it is as short as possible and that it operates without locking. The following processing happens when an interrupt is received and the LRNG is triggered:

1. A high-resolution time stamp is obtained using the service `random_get_entropy` kernel function. Although that function returns a 64-bit integer, only the bottom 32 bits, i.e. the fast moving bits, are used for further processing. Entropy is contained in the variations of the time of events and its time delta variations. Figure 2.1 depicts the 32-bit variable holding the time stamp.

2. A health test is now performed with the "stuck" test. That health test calculates the first, second and third discrete derivative of the time stamp of the interrupt event compared to the previous interrupt events. If one of those is zero, the recorded bit from step 3 is considered stuck. If a bit is found to be stuck, the processing of the IRQ event terminates, i.e. the entropy pool will not be touched. Entropy is found in the variations of the time deltas of interrupt event occurrences. Thus, the stuck test ensures that:

(a) variations exist in the time deltas,

(b) variations of time deltas do not have a simple repeating pattern, and

(c) variations do not have a linearly changing patterns (e.g. 1 - 2 - 4 - 7 - 11 - 16).

3. The time stamp value is a 32-bit integer is now processed with an LFSR to mix the data into the entropy pool. The LFSR operation performs the following steps:

   (a) The processing of the input data is performed either byte-wise or word-wise. The entropy pool is processed word-wise with a word size of 32 bits.

   (b) In case of a byte-wise processing of the input data, every byte is padded with zeroes to fill a 32 bit integer,

   (c) The 32 bit integer is rolled left by a value driven by variable that is increased by 7 with a wrap-around handling at 32 before processing one byte. The idea is that the input data is evenly mixed into the entropy pool. The used value of 7 ensures that the individual bits of the input data have an equal chance to move the bits within the entropy pool.

   (d) The resulting 32 bit integer is processed with the LFSR polynomial and inserted into the current word of the entropy pool. The LFSR polynomial is primitive, derived from a table of LFSR polynomials of various sizes. Note, however, that the taps in that document have to be reduced by one for the LRNG operation as the taps are used as an index into an array of words which starts at zero.

   (e) The pointer to the current word is increased by a prime number to point to the next word. The idea to use a prime number for the increment is to eliminate any potential dependencies of the taps in the LFSR. Note, for some LFSR polynomials, the taps are very close together.

4. If equal or more than /proc/sys/kernel/random/read_wakeup_threshold healthy bits are received, the wait queue where readers wait for entropy is woken up. Note, to limit the amount of wakeup calls if the entropy pool is full, a wakeup call is only performed after receiving 32 interrupt events. The reason is that the entropy pool can be read in 32-bit increments only anyway.

5. If the primary DRNG is fully seeded, the processing stops. This implies that only during boot time the next step is triggered. At runtime, the interrupt noise source will not trigger a reseeding of the primary DRNG.

6. If less than `LRNG_IRQ_ENTROPY_BITS` healthy bits are received, the processing of the LRNG interrupt callback terminates. This value denominates the number of healthy bits that must be collected to assume this bit string has 256 bits of entropy. That value is set to a default value of 256 (interrupts). Section 2.3.1 explains this default value. Note, during boot time, this value is set to 128 bits of entropy.

7. Otherwise, the LRNG triggers a kernel work queue to perform a seeding operation discussed in section 2.5.

The entropy collection mechanism is available right from the beginning of the kernel. Thus even the very first interrupts processed by the kernel are recorded by the aforementioned logic.

In case the underlying system does not support a high-resolution time stamp, step 2 in the aforementioned list is changed to fold the following 32 bit values each into one bit and XOR all of those bits to obtain one final bit:

- IRQ number,
- High 32 bits of the instruction pointer,
- Low 32 bits of the instruction pointer,
- A 32 bit value obtained from a register value – the LRNG iterates through all registers present on the system.

### 2.3.1 Entropy Amount of Interrupts

The question now arises, how much entropy is generated with the interrupt noise source. The current implementation implicitly assumes one bit of entropy per time stamp obtained for one interrupt[16].

When the high-resolution time stamp is not present, the entropy contents assumed with each received interrupt is divided by the factor defined with `LRNG_IRQ_OVERSAMPLING_FACTOR`. With different words, the LRNG needs to collect `LRNG_IRQ_OVERSAMPLING_FACTOR` more interrupts to reach the same level of entropy than when having the high-resolution time stamp. That value is set to 10 as a default.

## 2.4 HID Event Processing

The LRNG picks up the HID event numbers of each HID event such as a key press or a mouse movement by implementing the `add_input_randomness` function. The following processing is performed when receiving an event:

1. The LRNG checks if the received event value is identical to the previous one. If so, the event is discarded to prevent auto-repeats and the like to be processed.

2. The event values are processed with the LFSR used for interrupts as well. The LFSR therefore injects the HID event information into the entropy pool.

The LRNG does not credit any entropy for the HID event values.

[16] That value can be changed if the default is considered inappropriate. At compile time, the value of `LRNG_IRQ_ENTROPY_BYTES` can be altered. This value defines the number of interrupts that must be received to obtain an entropy content equal to the security strength of the used DRNG.

## 2.5 Primary DRNG Seeding Operation

The seeding operation obtains random data from the entropy pool. In addition it pulls data from the fast entropy sources of the CPU noise source if available. As these noise sources provide data on demand, care must be taken that they do not monopolize the interrupt noise source. This is ensured with the design choice to pull data from these fast noise sources at the time the interrupt noise source has sufficient entropy.

The (re)seeding logic tries to obtain 256 bits of entropy from the noise sources. However, if less entropy can only be delivered, the primary DRNG is able to handle this situation.

The entropy pool has a size of 128 32-bit words. The value of 128 words is chosen arbitrarily and can be changed to any other size provided another LFSR polynomial is provided.

For efficiency reasons, the seeding operation uses a seed buffer depicted in figure 2.1 that is one block of 256 bits and a second block equal to the digest size of the hash used to read the entropy pool. The first block is filled with data from the hashed data from the entropy pool. That buffer receives as much data from the hash operation as entropy can be pulled from the entropy pool. In the worst case when no new interrupts are received a zero buffer will be injected into the DRNG.

The second 256-bit blocks are dedicated the fast noise sources and is filled with data from those noise sources – i.e. RDRAND. If the fast noise sources is deactivated, its 256 bit block is zero and zero bits of entropy is assumed for this block. The fast noise source is only pulled if either entropy was obtained from the slow noise sources or the data is intended for the secondary DRNG. The reason is that the fast noise sources can dominate the slow noise sources when much entropic data is required. This scenario is prevented for /dev/random.

When reading the interrupt entropy pool, the entire entropy pool is hashed. The result of the hash is injected back into the entropy pool using the LFSR described in section 2.3. During the hashing, the LRNG processes the amount of entropy assumed to be present in the entropy pool. If the entropy is smaller than the hash size, the digest returned to the caller for the primary DRNG is truncated to a size equal to the amount of entropy that is present in the entropy pool. This operation is followed by reducing the assumed entropy in the pool by the amount returned by the hash operation.

Finally, also a 32 bit time stamp indicating the time of the request is mixed into the primary DRNG. That time stamp, however, is not assumed to have entropy and is only there to further stir the state of the DRNG.

During boot time, the number of required interrupts for seeding the DRNG is first set to an emergency threshold of one word, i.e. 32 bits. This is followed by setting the threshold value to deliver at least 128 bits of entropy. At that entropy threshold, the DRNG is considered "minimally" seeded – the value of 128 bits covers the minimum entropy requirement specified in SP800-131A ([5]) and complies with the minimum entropy requirement from BSI TR-02102 ([6]) as well. When reaching the minimal seed level, the threshold for the number of required interrupts for seeding the DRNG is set to `LRNG_IRQ_ENTROPY_BITS` to allow the DRNG to be seeded with full security strength.

### 2.5.1 Entropy of CPU Noise Source

The noise source of the CPU is assumed to have one 32th of the generated data size – 8 bits of entropy. The reason for that conservative estimate is that the design and implementation of those noise sources is not commonly known and reviewable. The entropy value can be altered by writing an integer into /sys/module/lrng/parameters/archrandom or by setting the kernel command line option of `lrng.archrandom`.

### 2.5.2 Entropy of CPU Jitter RNG Noise Source

The CPU Jitter RNG noise source is assumed provide 16th bit of entropy per generated data bit. Albeit studies have shown that significant more entropy is provided by this noise source, a conservative estimate is applied.

The entropy value can be altered by writing an integer into /sys/module/lrng/parameters/jitterrng or by setting the kernel command line option of `lrng.jitterrng`.

## 2.6 Secondary DRNG Seeding Operation

The secondary DRNG is seeded from the primary DRNG. Before obtaining random data from the primary DRNG, the LRNG tries to reseed the primary DRNG with 256 bits of entropy. That is followed by a generation of random numbers equal to the entropy content in the primary DRNG.

The secondary DRNG seeding operation may trigger a reseeding operation of the primary DRNG. In this case, the reseeding operation of the primary DRNG will always leave an emergency level of entropy in the entropy pool to be used exclusively for the primary DRNG. In addition, the seeding operation of the primary DRNG when triggered by the secondary DRNG will either obtain full 256 bits of entropy or nothing. This approach shall cover the concerns outlined in section 2.1.4.

In the worst case, the primary DRNG is unable to return any random numbers as it is not seeded with any entropy. Yet, the secondary DRNG will continue to operate considering that it was seeded with 256 bits of entropy during boot time.

## 2.7 LRNG-external Noise Sources

The LRNG also supports obtaining entropy from the following noise sources that are external to the LRNG. The buffers with random data provided by these noise sources are sent directly to the primary DRNG by invoking the DRNG's update function.

### 2.7.1 Kernel Hardware Random Number Generator Drivers

Drivers hooking into the kernel HW-random framework can inject entropy directly into the DRNG. Those drivers provide a buffer to the primary DRNG and an entropy estimate in bits. The primary DRNG uses the given size of entropy at face value. The interface function of `add_hwgenerator_randomness` is offered by the LRNG.

The amount of entropy injected into the primary DRNG is recorded to allow subsequent calls to read from the primary DRNG without additional reseeding.

Note: it is meaningless to inject more than the DRNG's security strength of data into the primary DRNG at once.

### 2.7.2 Injecting Data From User Space

User space can take the following actions to inject data into the DRNG:

- When writing data into /dev/random or /dev/urandom, the data is used to re-seed the primary DRNG and triggers a reseed of the secondary DRNGs at the time the next random number is about to be generated. The LRNG assumes it has zero bits of entropy.
- When using the privileged IOCTL of `RNDADDENTROPY` with /dev/random, the caller can inject entropic data into the primary DRNG and define the amount of entropy associated with that data.

Just like with the hardware RNGs, it is meaningless to inject more than the security strength of the DRNG at once.

User space may obtain the DRNG security strength size by reading /proc/sys/kernel/random/drng_security_strength.

## 2.8 DRBG

If the SP800-90A DRBG implementation is used, the default DRBG used by the LRNG is the CTR DRBG with AES-256. The reason for the choice of a CTR DRBG is its speed. The source code allows the use of other types of DRBG by simply defining a DRBG reference using the kernel crypto API DRBG string – see the top part of the source code for examples covering all types of DRBG.

Both the primary and secondary DRBG are always instantiated with the same DRNG type.

The implementation of the DRBG is taken from the Linux kernel crypto API. The use of the kernel crypto API to provide the cipher primitives allows using assembler or even hardware-accelerator backed cipher primitives. Such support should relieve the CPU from processing the cryptographic operation as much as possible.

The input with the seed and re-seed of the DRBG has been explained above and does not need to be re-iterated here. Mathematically speaking, the seed and re-seed data obtained from the noise sources and the LRNG external sources are mixed into the DRBG using the DRBG "update" function as defined by SP800-90A.

The DRBG generates output with the DRBG "generate" function that is specified in SP800-90A. The DRBG used to generate two types of output that are discussed in the following subsections.

### 2.8.1 /dev/urandom and `get_random_bytes`

Users that want to obtain data via the /dev/urandom user space interface or the `get_random_bytes` in-kernel API are delivered data that is obtained from the secondary DRNG "generate" function. I.e. the secondary DRNG generates the requested random numbers on demand.

Data requests on either interface is segmented into blocks of maximum 4096 bytes. For each block, the DRNG "generate" function is invoked individually. According to SP800-90A, the maximum numbers of bytes per DRBG "generate" request is $2^{19}$ bits or $2^{16}$bytes which is significantly more than enforced by the LRNG.

In addition to the slicing of the requests into blocks, the LRNG maintains a counter for the number of DRNG "generate" requests since the last reseed. According to SP800-90A, the number of allowed requests before a forceful reseed is $2^{48}$ – a number that is very high. The LRNG uses a much more conservative threshold of $2^{20}$requests as a maximum. When that threshold is reached, the secondary DRBG will be reseeded by using the operation documented in section 2.5 before the next DRNG "generate" operation commences.

The handling of the reseed threshold as well as the capping of the amount of random numbers generated with one DRNG "generate" operation ensures that the DRNG is operated compliant to all constraints in SP800-90A.

The reseed operation for /dev/urandom will drain the entropy pool down to a level where `LRNG_EMERG_POOLSIZE` interrupt events are still left in the pool. The goal is that even during heavy use of /dev/urandom, some emergency entropy is left for /dev/random. Note, before the DRNG is fully seeded, the `LRNG_EMERG_POOLSIZE` threshold is not enforced.

### 2.8.2 /dev/random

The random numbers to be generated for /dev/random are defined to have a special property: each bit of the random number produced for /dev/random is generated from one bit of entropy using the primary DRNG.

Naturally the amount of entropy the DRNG can hold is defined by the DRNG's security strength. For example, an HMAC SHA-256 DRNG or the ChaCha20-based DRNG can only hold 256 bits of entropy. Seeding that DRNG with more entropy without pulling random numbers from it will not increase the entropy level in that DRNG.

If the DRNG reseed request cannot be fulfilled due to the lack of available entropy, the caller is blocked until sufficient entropy has been collected or the DRNG has been reseeded with an entropy equal or larger to the security strength by external noise sources.

## 2.9 ChaCha20 DRNG

If the kernel crypto API support and the SP800-90A DRBG is not desired, the LRNG uses the standalone C implementations for ChaCha20 to provide a DRNG. In addition, the standalone SHA-1 C implementation is used to read the entropy pool.

The ChaCha20 DRNG is implemented with the components discussed in the following section. All of those components rest on a state defined by [11], section 2.3.

### 2.9.1 State Update Function

The state update function's purpose is to update the state of the ChaCha20 DRNG. That is achieved by

1. generating one output block of ChaCha20,

2. partition the generated ChaCha20 block into two key-sized chunks,

3. and XOR both chunks with the key part of the ChaCha20 state.

In addition, the nonce part of the state is incremented by one to ensure the uniqueness requirement of [11] chapter 4.

### 2.9.2 Seeding Operation

The seeding operation processes a seed of arbitrary lengths. The seed is segmented into ChaCha20 key size chunks which are sequentially processed by the following steps:

1. The key-size seed chunk is XORed into the ChaCha20 key location of the state.

2. This operation is followed by invoking the state update function.

3. Repeat the previous steps for all unprocessed key-sized seed chunks.

If the last seed chunk is smaller than the ChaCha20 key size, only the available bytes of the seed are XORed into the key location. This is logically equivalent to padding the right side of the seed with zeroes until that block is equal in size to the ChaCha20 key.

The invocation of the state update function is intended to eliminate any potentially existing dependencies between the seed chunks.

### 2.9.3 Generate Operation

The random numbers from the ChaCha20 DRNG are the data stream produced by ChaCha20, i.e. without the final XOR of the data stream with plaintext. Thus, the DRNG generate function simply invokes the ChaCha20 to produce the data stream as often as needed to produce the requested number of random bytes.

After the conclusion of the generate operation, the state update function is invoked to ensure enhanced backtracking resistance of the ChaCha20 state that was used to generate the random numbers.

## 2.10 PRNG Registered with Linux Kernel Crypto API

The LRNG supports an arbitrary PRNG registered with the Linux kernel crypto API, provided its seed size is either 32 bytes, 48 bytes or 64 bytes. To bring the seed data to be injected into the PRNG into the correct length, SHA-256, SHA-384 or SHA-512 is used, respectively.

## 2.11 getrandom in Atomic Contexts

The in-kernel API call of getrandom may be called in atomic context such as interrupts or spin locks. On the other hand, the kernel crypto API may sleep during the cipher operations used for the SP800-90A DRBG or the kernel crypto API PRNGs. The sleep would violate atomic operations.

This issue is solved in the LRNG with the following approach: The boot-time secondary DRNG provided with the ChaCha20 DRNG and a compile-time allocated memory for its context will never be released even when switching to another PRNG. The ChaCha20 DRNG can be used in atomic contexts because it never causes operations that violate atomic operations.

When switching the DRNG from ChaCha20 to another implementation, the ChaCha20 DRNG state of the secondary ChaCha20 DRNG is left to continue serving as a random number generator in atomic contexts. I.e. when the caller of getrandom is detected to be operating in an atomic context, the still present ChaCha20 DRNG is used to serve that request instead of the current secondary DRNG.

The seeding operation of the "atomic DRNG", however, cannot be triggered while getrandom is invoked, because the primary DRNG that it may seed from could be switched to the kernel crypto API and thus could sleep. To circumvent this issue, the seeding of the atomic DRNG is performed when a secondary DRNG is seeded. After the secondary DRNG is seeded and the atomic DRNG is in need of reseeding based on the reseed threshold, the time since last reseeding or a forced reseed, a random number is generated from that secondary DRNG and injected into the atomic DRNG.

## 2.12 Comparison With RNG Design Recommendations

### 2.12.1 SP800-90C

The specification of SP800-90C as provided in [2] and recently updated with [4] defines construction methods to design non-deterministic as well as deterministic RNGs. The specification defines different types of RNGs where the following mapping to the LRNG applies:

- The output of the /dev/random device complies with the Non-deterministic Random Bit Generator (NRBG) definition specified in section 5.6 of [4]. The reason is the use of an approved DRBG that is fed by an entropy source, if an SP800-90A DRBG is used. This DRBG is capable of providing output with "full entropy" when the caller applies the process described in section 9.4.2 [4] [17]. Although the entropy source is not always "live" as referenced in [4], the generation of output data stops until fresh entropy is received. If the entropy source(s) enter an error state – for example, the stuck test for the interrupt noise source always indicates stuck bits – and the entropy sources are incapable of generating entropy, the output of /dev/random stops which is considered to be the error state as defined in section 5.3 of [4].

  The primary DRBG operates as a DRBG with prediction resistance as defined in section 8.8 of [1] with one important difference: instead of requiring that the primary DRBG is always seeded with entropy equal to the full security strength of the DRBG, the primary DRBG keeps track on how much entropy was provided with a seed and delivers only that amount

[17] The primary DRBG generates at most as many bits of random numbers as the DRBG was seeded with and allows the caller to obtain "full entropy" random numbers as defined in section 9.4.2 [4]. As discussed below, the "full entropy" definition of section 5.2 and hence the process specified in 9.4.2 from [4] are not applied for reseeding the secondary DRBG.

of random data equal to the amount of entropy it was seeded with before the given generate operation.

- The output of the /dev/urandom device and the `get_random_bytes` kernel function is a DRBG without prediction resistance as allowed in chapter 4 of [4]. The reseed threshold, however, is significantly lower than specified with SP800-90A in [1]. In addition to a threshold regarding the amount of generated random data, the secondary DRBG also employs a time-based reseeding threshold to ensure that the DRBG is reseeded in a reasonable amount of time.

The requirements of the security of an RNG defined in section 4.1 of [4] are considered to be covered as follows:

1. The entropy source of the interrupt noise source complies with SP800-90B [3] as assessed above. For the CPU noise sources, no statement can be made as no access to the design and implementations are given.

2. The DRBG is designed according to SP800-90A and has received even FIPS 140-2 certification.

3. The primary DRBG is instantiated using input from the noise sources whereas the secondary DRBG is instantiated from the primary DRBG.

4. The LRNG is implemented entirely within the Linux kernel which implies that its entire state is protected from access by untrusted entities.

5. Data fetched from the noise sources always contains data with fresh, yet unused entropy. It may be possible that the entropy gathered from the noise sources cannot deliver as many entropic bits as requested. The primary DRBG tracks the amount of entropy gathered from the noise source to ensure that it only returns at most as much random data as the DRBG received in form of entropy.

According to section 5.2 [4], full entropy is defined as a random number generated by a DRBG that contains entropy of half of the random number size. The primary DRBG acting as NRBG is capable of complying with this definition at the request of the caller – i.e. the caller assumes that the obtained data from the primary DRBG has an entropy content that is half the size of the generated random numbers. The process of seeding another random number generator from the primary DRBG with the definition of "full entropy" applied is described in section 9.4.2 [4].

However, this full entropy definition is not applied for seeding the secondary DRBG. This means that process is described in section 9.4.2 of [4] is not used to seed the secondary DRBG. Various cryptographers, namely mathematicians from the German BSI, consider such compression factor as irrelevant. SP800-90C is yet in draft state and many other random number generators are implemented such that the amount of entropy injected into the DRBG allows an equal amount of random data to be extracted and yet consider that this data has full entropy content. If the SP800-90C full entropy definition shall be enforced, the reseeding operation of the secondary DRBG in `lrng_sdrng_seed` requires calling of the primary DRBG gathering function twice and assume that

the resulting bit string only contains an entropy content that is half of the data size of the returned random numbers.

As required in chapter 4 [3] and chapter 5 [4], the interrupt noise source implemented by the LRNG is subject to a health test. This health test is implemented with the stuck test.

Chapter 7 [4] specifies pseudo-code interfaces for the DRBG and NRBG where the LRNG only implements the "Generate_function". The "Instantiate_function" is not implemented as the LRNG implements and automatic instantiation. For the DRBG, a "Reseed_function" is implemented by allowing user space to write data to /dev/random or using the IOCTL to inject data into the DRBG as well as `add_hwgenerator_randomness`. The LRNG also implements the "GetEntropy" logic as defined in section 7.4 [4] where each noise source is accessed to obtain a bit stream and a value of the assessed entropy.

### 2.12.2 AIS 20 / 31

The German BSI defines construction methods of RNGs with AIS 20/31 [7]. In particular, this document defines different classes of RNGs in chapter 4.

The LRNG can be compared to the types of RNGs defined in AIS 20/31 as follows:

- The primary DRNG and is directly callable interface of /dev/random is an NTG.1 which uses one or more entropy sources. Each entropy source has an entropy estimation associated with it. The generation of the data for /dev/random considers this entropy estimate by reseeding the DRNG with a buffer holding an entropy amount equal or larger to the DRNG security strength. The state transition function $\varphi$ and output function $\psi$ are provided with the chosen DRNG. By using the primary DRNG to serve /dev/random, the LRNG ensures that each random number generated by the primary DRNG must be backed by an equal amount of entropy that seeded the DRNG. Hence, the data derived from /dev/random is backed by information theoretical entropy.

- The /dev/urandom and the in-kernel `get_random_bytes` function is a DRG.4. It uses a DRNG for the state transition function $\varphi$ and output function $\psi$ to ensure enhanced backward secrecy which is the prerequisite for a DRG.3. Due to the frequent reseed operation with fresh entropy and the use of an SP800-90A DRNG with its update function invoked during the generate operation, it also complies with the additional requirement of a DRG.4.

As required by AIS 20/31 [7] chapter 4, a health test of the noise source is implemented with the stuck test.

### 2.12.3 FIPS 140-2

The FIPS 140-2 standard refers to SP800-90A and SP800-90B regarding approved DRBGs and noise sources. However, one aspect must be enforced by the noise source: the continuous random number generator test (CRNGT). This test is detailed in FIPS 140-2 IG 9.8.

The continuous random number generator test requires that a comparison test must be implemented to identify a stuck test. It requires that the previous

data point is not identical to the current data point. When using integers, this requirement now means that both integers are not identical. That requirement is identical to calculating a difference of both integers and ensuring that it is not zero. If it is zero, the obtained data point is to be considered stuck.

Such test is implicitly implemented with the stuck test:

- The LRNG interrupt noise source receives a notification that an interrupt is received by the Linux kernel. The LRNG obtains a time stamp of that event.
- The LRNG implements the stuck test which calculates the first, second and third derivative of the time of the event. For the raw noise data of time stamps the CRNGT is enforced by calculating the the first derivative of the time, i.e. the delta time stamps. When that result is zero, it declares the obtained data point as stuck and does not treat it as having any entropy.

As discussed in section 2.3, a stuck time stamp is mixed into the pool but does not contribute to entropy. In the worst case of always collecting stuck measurements, no entropy is generated and delivered to the DRNG.

When the kernel is booted in FIPS mode – i.e. booted with `fips=1` – the LRNG causes the kernel to `panic` after reaching 4 back-to-back stuck tests. The reason for choosing the value of 3 allowed back-to-back stuck values and `panic`ing with the 4th stuck value is found in the minimum entropy value depicted in 3.2 which identifies the minimum entropy of the time delta, i.e. the value that defines the entropy for the interrupt noise source. As this value is about 11 bits, using this value with the formula in IG 9.8 results in allowing a threshold of about 3.

## 2.13 LRNG External Interfaces

The following LRNG interfaces are provided:

`add_interrupt_randomness` This function is to be hooked into the interrupt handler to trigger the LRNG interrupt noise source operation.

`add_input_randomness` This function is called by the HID layer to stir the entropy pool with HID event values.

`get_random_bytes` In-kernel equivalent to /dev/urandom.

`get_random_bytes_arch` In-kernel service function to safely call CPU noise sources directly and ensure that the LRNG is used as a fallback if the CPU noise source is not available.

`add_hwgenerator_randomness` Function for the HW RNG framework to fill the LRNG with entropy.

`add_random_ready_callback` Register a callback function that is invoked when the DRNG is fully seeded.

`del_random_ready_callback` Delete the registered callback.

`wait_for_random_bytes` With this function, a synchronous wait until the DRNG is fully seeded is implemented inside the kernel.

**/dev/random** User space interface to provide random data with full entropy – read function may block if insufficient entropy is available.

**/dev/urandom** User space interface to provide random data from a constantly reseeded DRNG – the read function will generate random data on demand. Note, the buffer size of the read requests should be as large as possible, up to 4096 bits to provide a fast operation. See table 6 for an indication of how important that factor is.

**/proc/sys/kernel/random/poolsize** Size of the entropy pool in bits.

**/proc/sys/kernel/random/entropy_avail** Number of interrupt events mixed into the entropy pool.

**/proc/sys/kernel/random/read_wakeup_threshold** Threshold that when reached by the entropy_avail value triggers a wakeup of readers. In addition, a wakeup of the readers is triggered when the DRNG is (re)seeded with security strength bits of entropy. The minimum allowed to be set is equal to the minimum amount of entropic bits that can be obtained with one read of the entropy pool, i.e. 32 bits.

**/proc/sys/kernel/random/write_wakeup_threshold** When entropy_avail falls below that threshold, suppliers of entropy are woken up.

**/proc/sys/kernel/random/boot_id** Unique UUID generated during boot.

**/proc/sys/kernel/random/uuid** Unique UUID that is re-generated during each request.

**/proc/sys/kernel/random/urandom_min_reseed_secs** Number of seconds after which the secondary DRNG will be reseeded. The default is 600 seconds. Note, this value can be set to any positive integer, including zero. When setting this value to zero, the secondary DRNG tries to reseed from the primary DRNG before every generate request. I.e. the secondary DRNG in this case acts like a DRNG with prediction resistance enabled as defined in [1].

**/proc/sys/kernel/random/drng_fully_seeded** Boolean indicating whether the DRNG is fully seeded with entropy equal to the DRNG security strength.

**/proc/sys/kernel/random/drng_minimally_seeded** Boolean indicating whether the DRNG is seeded the minimum entropy of 128 bits.

**/proc/sys/kernel/random/lrng_type** String referencing the DRNG type and the security strength of the DRNG.

**/proc/sys/kernel/random/drng_security_strength** Security strength of the used DRNG in bytes. This value is important for suppliers of entropy: providing more entropy will not make the /dev/random operation faster as the primary DRNG will only be able to store that amount of entropy. In addition, the secondary DRNG requires DRNG security strength random numbers from the primary DRNG during reseeding. Hence, it is suggested to provide random data with entropy that matches exactly the

given DRNG security strength to serve both /dev/random and /dev/urandom.

IOCTLs are implemented as documented in random(4).

# 3 LRNG Testing

After the discussion of the design of the LRNG, the LRNG must now show that it is up to the task. Testing is performed for the following areas:

- Tests to compare the LRNG with the legacy /dev/random are conducted to analyze whether the LRNG brings benefits over the legacy implementation.
- Statistical tests of the LRNG data are conducted. These statistical tests focus on the raw entropic data from the interrupts and reach to the assessment of output of the DRNG to demonstrate that the DRNG is correctly used.

Before presenting the test results, the first subsection discusses the used statistical tools and the test approach.

## 3.1 Statistical Tools

The statistical analysis of the various data sets is performed with different tools that are explained in the following.

The first tool used is `ent` that is accessible at the ent homepage. That tool calculates the following statistical values:

- Shannon Entropy value: given the fact that we subsequently calculate a minimum entropy value, this value is considered to be secondary to the assessment. Note, its result is perfect in almost all cases and thus not really helpful.
- Compression rate: this value is also considered secondary as often a perfect compression rate of zero was displayed by `ent`, but `bzip2` could compress the data set. Hence, that value is not considered during the assessment either and is replaced with an invocation of `bzip2` if deemed relevant.
- Chi-Square value: this value and its rating regarding the confidence interval is the most important result of the `ent` tool. A passing value is a necessary for determining that a data set is considered random. However, a good value does not automatically imply that the data set shows no other weaknesses. The Chi-Square value therefore is used as a "smoke test" in all circumstances to identify whether a data set is non-random to begin with.
- The arithmetic mean value is a good indicator whether any skews are present. If that value is not right in the center of the distribution, the Chi-Square test will fail too. Hence, this value may inform about why a Chi-Square test fails.

- The Monte Carlo Test of PI seems to be less relevant. Its result correlates with the result of Chi-Square and the arithmetic mean value.
- The serial correlation coefficient is another important value indicating the correlation of the data set values. Similarly to the arithmetic mean value, it provides hints why a Chi-Square test fails.

`ent` allows the calculation of the mentioned values to be performed with a bitwise and bytewise treatment of the input data set. All data sets used here are considered be independent and identically distributed (IID) which implies that both the bitwise and the bytewise calculations should show equal results. Hence, the following testing analyzes the data sets always bitwise and bytewise. When `ent` is used on data sets that are the output of a cryptographic whitening function such as the DRNG, its result is used to determine that the implementation does not introduce any coding errors similarly to CVE:2013-4345, because a statistical test of such data must by definition be perfect.

In addition to `ent`, the statistical tests defined by SP800-90B ([3]) are used to calculate various minimum entropy values. [3] provides full definitions of the tests which are not specified here again apart from the following list of conducted statistical tests resulting in a minimum entropy calculation:

- Bins test
- Collision test
- Compression test
- Collection test
- Markov test

The assessment below uses these tests to determine the minimum entropy of the raw noise before it is processed with the DRNG. As the output of the DRNG is a cryptographically whitened data set, any calculation of the minimum entropy value for these data sets must by definition be perfect and is therefore skipped.

Furthermore, the statistical tests defined by SP800-22rev1a ([12]) are applied using the test tool provided at the NIST website. The document of [12] defines all tests and provides a rationale about how to interpret the results. Hence, the documentation below only notes whether these tests pass or fail. SP800-22rev1a performs the following tests:

- Frequency test
- Block Frequency test
- Cumulative Sums test
- Runs test
- Longest Run test
- Rank test
- Fast Fourier Transformation test
- Non Overlapping Template test

- Overlapping Template test
- Universal test
- Approximate Entropy test
- Random Excursions test
- Random Excursions Variant test
- Serial test
- Linear Complexity test

The German BSI AIS 20/31 specified in [7] defines a set of statistical tests called "test procedure A" consisting of the following tests:

- Monobit Test
- Poker Test
- Runs Test
- Long Runs Test
- Autocorrelation Test

This test procedure is defined in section 2.4.4.1 [7]. The tests are fully specified in that document and are not re-iterated here. The test procedure A requires at least 5 MB of data.

## 3.2 Test Approach

The testing requires data that is maintained within the kernel and does not have an architected interface to access or read that data. For accessing such information and exporting it to user space for further assessment, SystemTap is used. The following SystemTap scripts are used:

- The SystemTap script `lrng_irq_raw_time.stp` is hooked into the LRNG interrupt handler function. This script reads out the time stamp value generated for an interrupt and that is folded into one bit. This time stamp is the raw entropy source for the LRNG.
- The script `lrng_irq_raw_entropy.stp` is hooked into the DRNG reseed function and obtains a snapshot of the interrupt entropy pool. The script is to be used such that only reseed triggers from the interrupt handler. This ensures that the entropy pool is filled with data from 256 interrupts and thus allows measuring the statistical properties of the data used to seed the DRNG.

The output functions of the LRNG are binary compatible to the interfaces of the legacy /dev/random. Therefore, to test the LRNG operation, the LRNG API must be hooked into the kernel – which is implemented with a very simple approach: The LRNG interface functions are added to the legacy /dev/random code. The following code snippets shows the changes to the legacy /dev/random code:

Listing 1: drivers/char/random.c

```
+extern void lrng_get_random_bytes(u8 *outbuf, u32 outbuflen);
 void get_random_bytes(void *buf, int nbytes)
 {
+       return lrng_get_random_bytes(buf, nbytes);
...
+extern  ssize_t
+lrng_full_read(struct file *file, char __user *buf, size_t nbytes, loff_t *ppos);
 static ssize_t
 random_read(struct file *file, char __user *buf, size_t nbytes, loff_t *ppos)
 {
+       return lrng_full_read(file, buf, nbytes, ppos);
...
+extern  ssize_t
+lrng_drbg_read(struct file *file, char __user *buf, size_t nbytes, loff_t *ppos);
 static ssize_t
 urandom_read(struct file *file, char __user *buf, size_t nbytes, loff_t *ppos)
 {
@@ -1463,6 +1470,7 @@ urandom_read(struct file *file, char __user *buf, size_t nbytes, loff_t *ppos)
                printk_once(KERN_NOTICE "random: %s urandom read "
                            "with %d bits of entropy available\n",
                            current->comm, nonblocking_pool.entropy_total);
+       return lrng_drbg_read(file, buf, nbytes, ppos);
...
+extern unsigned int lrng_full_poll(struct file *file, poll_table * wait);
 static unsigned int
 random_poll(struct file *file, poll_table * wait)
 {
        unsigned int mask;
+       return lrng_full_poll(file, wait);
...
+extern ssize_t lrng_drbg_write(struct file *file, const char __user *buffer,
+                               size_t count, loff_t *ppos);
 static ssize_t random_write(struct file *file, const char __user *buffer,
                            size_t count, loff_t *ppos)
 {
        size_t ret;
+       return lrng_drbg_write(file, buffer, count, ppos);
...
+extern long lrng_ioctl(struct file *f, unsigned int cmd, unsigned long arg);
 static long random_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
 {
        int size, ent_count;
        int __user *p = (int __user *)arg;
        int retval;
+       return lrng_ioctl(f, cmd, arg);
...
+extern int lrng_fasync(int fd, struct file *filp, int on);
 static int random_fasync(int fd, struct file *filp, int on)
 {
+       return lrng_fasync(fd, filp, on);
...
+extern ssize_t lrng_getrandom(char __user * buf, size_t count, unsigned int flags);
 SYSCALL_DEFINE3(getrandom, char __user *, buf, size_t, count,
                unsigned int, flags)
 {
+       return lrng_getrandom(buf, count, flags);
...
+extern void lrng_add_hwgenerator_randomness(const char *buffer, size_t count,
+                                            size_t entropy);
 void add_hwgenerator_randomness(const char *buffer, size_t count,
                                 size_t entropy)
 {
        struct entropy_store *poolp = &input_pool;
+       return lrng_add_hwgenerator_randomness(buffer, count, entropy);
```

Using the changes to the legacy /dev/random code, the LRNG is now directly used when accessing the interfaces of the legacy /dev/random. Yet, the legacy /dev/random operation to record and maintain entropy is still active.

All tests are executed within a KVM guest environment without graphical environment. The host is an Intel Core i7 Broadwell system.

## 3.3 LRNG Comparison With Legacy /dev/random

The first round of testing tries to compare the legacy /dev/random operation with the LRNG operation. All tests were executed on the test system operating as a KVM guest.

In addition, the same tests were executed on a Linux kernel operating on bare metal on an Intel Atom Z-530 processor. As this system is very slow and old, an indication of whether the LRNG shows different behavior on old and slow systems was obtained.

### 3.3.1 Time Until Fully Initialized

The legacy /dev/random implementation feeds all entropy directly into the `nonblocking_pool` until the kernel log message is recorded that the `nonblocking_pool` is initialized. Only after that point, entropy is fed into the `input_pool` allowing the seeding of the `blocking_pool` and thus generating data for /dev/random.

The DRNG also prints out a message when it is fully seeded. The following test lists these two kernel log messages including their time stamp.

As mentioned above, the DRNG uses different noise sources where only the interrupt noise source will always be present. Thus the test is first performed with all noise sources enabled followed by disabling the fast noise sources of CPU noise source.

Listing 2: Time until fully initialized -- LRNG using all noise sources

```
$ dmesg | grep "primary DRNG minimally seeded"
[    0.808266] lrng: primary DRNG minimally seeded
--- 0 sm@x86-64 ~ ------------------------------------------------------------
$ dmesg | grep "primary DRNG fully seeded"
[    1.063844] lrng: primary DRNG fully seeded
--- 0 sm@x86-64 ~ ------------------------------------------------------------
$ dmesg | grep "nonblocking"
[   20.932050] random: nonblocking pool is initialized
```

The test shows that the primary DRNG is minimally seeded 0.6 seconds after boot. This is around the time when the initramfs is started. The primary DRNG is fully seeded at 1 second after boot which is before systemd injects the legacy /dev/random seed file into /dev/random and before the initramfs terminates. As the secondary DRNG is immediately seeded from the primary DRNG at the time of reaching the minimally and fully seeding threshold, the aforementioned listing applies to the secondary DRNG too.

The legacy /dev/random's `nonblocking_pool` on the other hand is initialized with 128 bits of entropy at around 21 seconds after boot in this test round – other tests show that it may even be initialized after 30 seconds and more. By that time the complete boot process of the user space is already long completed.

The following test boots the kernel with the kernel command line options of `lrng.archrandom=0` to disable the fast noise sources.

Listing 3: Time until fully initialized -- LRNG using only interrupt noise source

```
$ cat /sys/module/lrng/parameters/archrandom
0
--- 0 sm@x86-64 ~ ------------------------------------------------------------
$ dmesg | grep "DRNG minimally seeded"
[    1.057013] lrng: primary DRNG minimally seeded
--- 0 sm@x86-64 ~ ------------------------------------------------------------
$ dmesg | grep "DRNG fully seeded"
[    1.309164] lrng: primary DRNG fully seeded
--- 0 sm@x86-64 ~ ------------------------------------------------------------
```

```
[    1.497105] lrng: inject 64 bytes with 0 bits of entropy into primary DRNG
[    1.497108] lrng: seeding secondary DRNG with 64 bytes
[    1.497133] lrng: inject 64 bytes with 0 bits of entropy into primary DRNG
[    1.497134] lrng: seeding secondary DRNG with 64 bytes
[    1.497156] lrng: inject 64 bytes with 0 bits of entropy into primary DRNG
[    1.497157] lrng: seeding secondary DRNG with 64 bytes
[    1.497181] lrng: inject 64 bytes with 0 bits of entropy into primary DRNG
[    1.497182] lrng: seeding secondary DRNG with 64 bytes
[    1.497225] lrng: inject 64 bytes with 0 bits of entropy into primary DRNG
[    1.497226] lrng: seeding secondary DRNG with 64 bytes
[    1.497249] lrng: inject 64 bytes with 0 bits of entropy into primary DRNG
[    1.497250] lrng: seeding secondary DRNG with 64 bytes
[    1.497272] lrng: inject 64 bytes with 0 bits of entropy into primary DRNG
[    1.497273] lrng: seeding secondary DRNG with 64 bytes
[    1.497328] lrng: inject 64 bytes with 0 bits of entropy into primary DRNG
[    1.497339] lrng: seeding secondary DRNG with 64 bytes
--- 0 sm@x86-64 ~ ---------------------------------------------------------------
$ dmesg | grep "nonblocking"
[   33.206682] random: nonblocking pool is initialized
```

Even when the fast noise sources are disabled, the DRNG is minimally and fully initialized at the time the initramfs started. Conversely, the `nonblocking_pool` is initialized again long after the boot process is completed.

During all testing, the DRNG was fully seeded before user space injected the seed data into /dev/random as mandated by the legacy /dev/random implementation. The time of user space injecting the seed data into /dev/random marks the point at which cryptographically relevant user space applications may be started.

As the DRNG is fully seeded at the time of initramfs, user space daemons requiring cryptographically strong random numbers are delivered such data.

### 3.3.2 Entropy Delivered To First User Space Caller

Another comparison can be made between the legacy /dev/random and the LRNG regarding the amount of entropy delivered to the first user space caller.

Again, for the LRNG, the first test is performed with all noise sources active and the second test having the fast noise sources disabled.

Listing 4: Entropy for first user space caller -- LRNG using all noise sources

```
[    0.546574] lrng: inject 68 bytes with 64 bits of entropy into primary DRNG
[    0.546576] lrng: primary DRNG initially seeded
[    0.546588] lrng: seeding secondary DRNG with 32 bytes
[    0.557907] random: systemd urandom read with 3 bits of entropy available
```

The first user space caller obtains random numbers from a DRNG seeded with 64 bits of entropy. The listing shows that in fact, the DRNG is seeded with 192 bits of entropy.

Listing 5: Entropy for first user space caller -- LRNG using only interrupt noise source

```
[    0.571599] lrng: inject 100 bytes with 56 bits of entropy into DRNG
[    0.571607] lrng: seeding secondary DRNG with 32 bytes
[    0.575029] random: systemd urandom read with 3 bits of entropy available
```

When the fast noise sources are disabled, the LRNG already obtained 56 bits of entropy from interrupts at the time the first user space caller. Note, the secondary DRNG is seeded with 32 bytes from the primary DRNG when the primary DRNG seed level is below the minimally seeded threshold. Yet, that random number contains the entropy amount the primary DRNG was seeded with, i.e. 56 bits in the given example.

### 3.3.3 Entropy In LRNG When `nonblocking_pool` Is Initialized

With the previous tests in mind, the question may be interesting on how much entropy the LRNG already obtained at the time the legacy /dev/random has its `nonblocking_pool` initialized.

The test is performed on an otherwise quiet system which implies that after boot, only very few operations with little interrupt activity but also little draw on entropy is ongoing. Hence, the DRNG is only seeded during boot which is visible with the long time delay between the last DRNG notification and the `nonblocking_pool` kernel log message. As already done for the previous tests, the LRNG is tested once with and once without fast noise sources.

Listing 6: Entropy in LRNG when `nonblocking_pool` is initialized -- LRNG using all noise sources

```
$ cat /proc/sys/kernel/random/entropy_avail
2036

[   25.852725] random: nonblocking pool is initialized
```

At the time the `nonblocking_pool` is initialized, the LRNG has already produced about 2000 bits of entropy to be used with /dev/random which means that at that time, LRNG requests at /dev/random can be satisfied for 250 bytes right away.

Listing 7: Entropy in LRNG when `nonblocking_pool` is initialized -- LRNG using only interrupt noise source

```
$ cat /proc/sys/kernel/random/entropy_avail
1862

[   32.912590] random: nonblocking pool is initialized
```

Even without the fast noise sources, the LRNG is able to produce 232 bytes of random data for /dev/random by the time the `nonblocking_pool` is initialized and the legacy /dev/random starts to fill its `input_pool` that can be drawn from to satisfy /dev/random requests.

## 3.4 Statistical Tests of LRNG

After showing the comparison between the operation of the legacy /dev/random and the LRNG, statistical tests of the quality of input and output data of the LRNG are done and analyzed.

The statistical tests are performed for each processing step of the LRNG:

1. raw time stamp obtained from an interrupt operation,
2. result of the LFSR operation to mix time stamps into the entropy pool,
3. analysis of the boot time variances,
4. output of the DRNG for /dev/random, and
5. output of the DRNG for /dev/urandom.

### 3.4.1 IRQ Noise Source – Time Stamp

Using the SystemTap script `lrng_irq_raw_time.stp` the raw time stamps for interrupts are recorded. Those time stamps are the input to the folding operation to fill the interrupt entropy pool[18].

The first assessment is performed on the raw time stamps obtained from the measurement. A histogram of those measurements is prepared where the ideal would be a rectangular distribution without any spikes. Such rectangular distribution identifies an equidistribution of the time stamps and is a prerequisite for determining whether the data can be considered to be white noise.

To obtain that amount of data – about 55 million time stamps – the following test approach is used: To generate time stamps, interrupts must be generated. The large number of interrupts is generated by using a ping flood from the host. Note, the test system is executed in a KVM guest. On the host system, a ping flood is generated to ping the IP address of the test machine. This operation should be considered as a worst case test as:

- the generation of interrupts is triggered externally from a potential adversary,
- the network latency introduced with routers or switches is eliminated to the greatest extent,
- each ping packet that triggers an interrupt is generated with the same time delay – i.e. the ping command uses a tight loop to generate these packets, and
- the potential adversary is able to generate large numbers of interrupts that shall serve as entropy.

It is to be noted that the hardware used for the testing has 2 physical cores with 2 hyper-threads each. The KVM is executed with 2 virtual CPUs occupying 2 hyper-threads and the host using the other 2 hyper-threads. This reduces the latency from the host Linux kernel scheduling as much as possible to again try to define a worst case testing approach.

The following command is used on the host for initiating the ping flood:

```
sudo ping -f <TEST-IP-ADDRESS>
```

The obtained data is converted into the histogram provided with figure 3.2. That figure shows in its legend the minimum entropy of the data set obtained with the following formula:

$$H_{min} = -log_2(p_{max})$$

with $p_{max}$ describing the maximum probability of an event – the time delta. In addition, it shows the Shannon Entropy obtained with the formula of:

$$H = -\sum_{i=1}^{N} p_i \cdot log_2(p_i)$$

[18]Note, this SystemTap script reads out a 64-bit CPU register which returns a 64-bit value. As we are only interested in the 32-bit value of this data cutting off the high 32 bits, the application `timetoint.c` is provided to convert the SystemTap output into 32-bit integers.

where $p_i$ denominates the probability of an event – the time delta – and $N$ represents the number of events.

The histogram contains a standard Gaussian distribution with the red dotted line using the mean and standard deviation of the collected data. The green vertical lines are the first and third quartile of the distribution. The median of the data set is given with the blue and the mean with the red vertical line.

Figure 3.1 presents the histogram for the raw time stamps. Note, this histogram is generated by using the first 5,000,000 time stamps of the test result. The reason for limiting the number of time stamps considered is due to the limits of R: on a 16 GB RAM system, R complained about insufficient memory when generating an SVG with more time stamps.

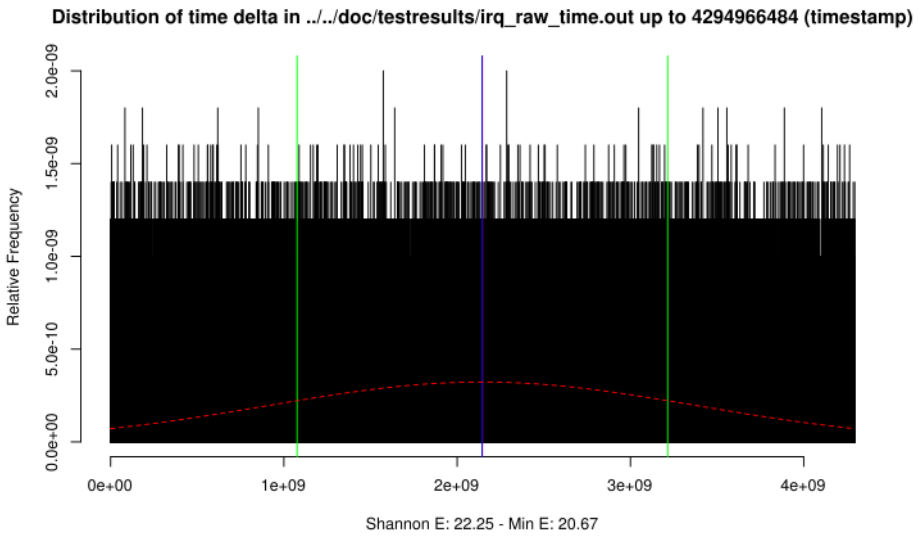


Figure 3.1: Histogram of interrupt time stamps

The result shows an almost perfect rectangular distribution. Naturally, it may happen that some time stamp values are recorded twice or three times which explains the few spikes. Furthermore, the use of only 10% of the original data set to draw that figure supports the case that a few spikes are seen. Yet the overall distribution type is an equidistribution as expected. The values for minimum and Shannon entropy for the 5 million time stamps must be viewed in light of the number of test records compared to the overall number space of $2^{32}$. The 5 million time stamps are about 1.5 times $2^{22}$. Hence, the Shannon entropy of 22.3 bits is very close to the theoretical entropy. The minimum entropy of 20.7 bits also reaches close to the maximum possible entropy. The entropy values for the full data set of 55 million – which is about 1.5 times $2^{25}$ – are about 25.7 bits of Shannon entropy and 23.7 bits of minimum entropy. These entropy values are yet again close to the theoretical maximum entropy. Using this finding, the conclusion can be drawn that when obtaining more than $2^{32}$ time stamp samples, the maximum possible entropy of 32 bits would nearly be reached for Shannon as well as minimum entropy. Hence, the figure and the resulting statistical numbers indicate that the distribution of the time stamp

values over the possible values between 0 to $2^{32}$ follows an equidistribution as expected.

In addition to calculating of the minimum entropy above, another supporting minimum entropy value is calculated based on the Markov model. The Markov model and its formulas is described in [3] section 9.3.5.3. The time stamps are 32-bit values. To apply the Markov test, 32-bit values are too big. Hence, the fast moving low bits of the time stamps are obtained from each time stamp and converted into a bit-stream. For the following test, the low four bits are used out of each time stamp to form the upper and lower nibble of a byte. The bytes obtained out of two time stamps are concatenated to form a byte stream that is processed with the Markov test. The conversion of the time stamp to the described byte stream is performed with the test application `timetomarkov.c`.

The following table lists the calculated minimum entropy using the Markov test in relationship to the chosen step size.

| Step Size | Markov Min Entropy |
|---|---|
| 1 | 3.92 |
| 2 | 7.84 |
| 3 | 11.76 |
| 4 | 15.67 |
| 5 | 19.59 |
| 6 | 23.50 |
| 7 | 27.42 |
| 8 | 31.34 |
| 9 | 35.25 |

Table 1: Markov test minimum entropy of interrupt time stamps

The calculated Markov minimal entropy for a given step size is always very close to the maximum possible entropy size which is the step size times 4 bit – the Markov minimum entropy for the different step sizes is always about 98% of the maximum possible entropy. Hence, the time stamps contain entropy that is very close to the maximum entropy possible.

The next analysis is performed with the time deltas. The absolute time stamps are monotonically increasing values. Hence, the entropy is not too well visible using this value. When calculating the first derivative of the time deltas, the uncertainty that delivers entropy is visible in the variations of those time deltas. The following histogram shows the distribution of the time delta variations of the recorded time stamps where again the upper 1 percentile is removed to eliminate large outliers – note that elimination serves as an assessment of a worst case because the outliers add more entropy to the LRNG. Figure 3.2 shows the histogram of the time deltas.

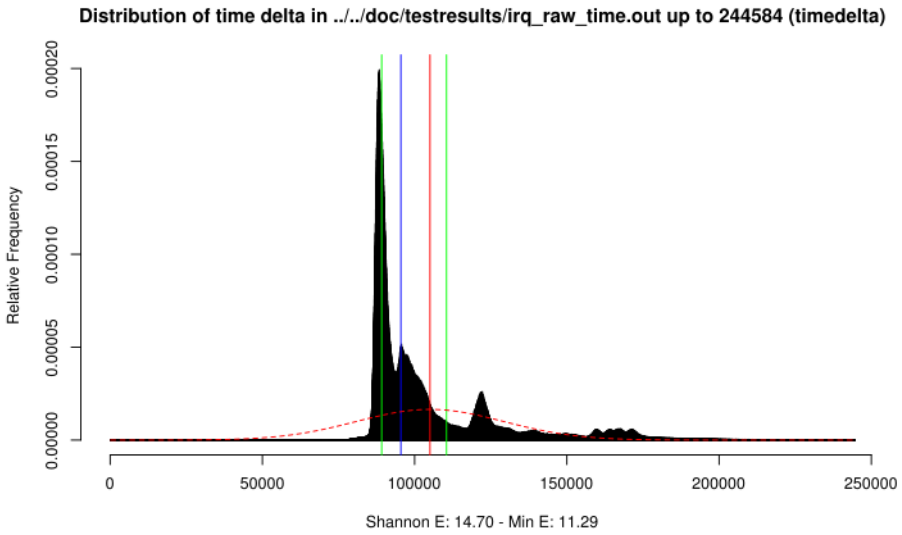


Figure 3.2: Histogram of interrupt time deltas

The histogram shows a wide spread of the time delta of the ping flood interrupts. The minimum entropy is more than 11 bits and the Shannon Entropy is more than 14 bits. As one measurement provides about one bit of entropy, the raw entropic data contains significantly more entropy than needed.

Again, the minimum entropy following the Markov model is calculated for the time deltas using the same consideration above: the four low bits of the time deltas are used for an analysis with the Markov model. The conversion of the time stamp to the time delta followed by obtaining the low 4 bits is performed with the test application `timedeltatomarkov.c`.

The following table lists the calculated minimum entropy using the Markov test in relationship to the chosen step size.

| Step Size | Markov Min Entropy |
|---|---|
| 1 | 3.92 |
| 2 | 7.84 |
| 3 | 11.75 |
| 4 | 15.67 |
| 5 | 19.58 |
| 6 | 23.49 |
| 7 | 27.41 |
| 8 | 31.32 |
| 9 | 35.23 |

Table 2: Markov test minimum entropy of interrupt time deltas

The calculated Markov minimal entropy is almost identical to the minimum entropy for the time stamps. It is always about 98% of the maximum possible entropy. Hence, the time stamps contain entropy that is very close to the

maximum entropy possible.

The testing for figure 3.2 was conducted in a KVM virtual machine. Additional tests on Microsoft Hyper-V, Oracle VirtualBox, VMWare ESXi, and Xen with and without para-virtualized device drivers show similar results. Though, one major finding was that the para-virtualized device drivers for Hyper-V use the VMBus interface to the host. VMBus, however, does not use the common Linux kernel interrupt handling code and per default would not trigger the LRNG interrupt callback. The results on Hyper-V were obtained with the legacy network device configuration – emulating a DEC network interface – as well as with the default network configuration using the VMBus. This test result was the key motivation for providing a patch to the VMBus handling code to trigger the LRNG interrupt collection function. The following table shows the Markov test minimum entropy for the additional virtual machine tests, each with step size 1.

| Virtual Machine Monitor | Markov Min Entropy |
|---|---|
| Hyper-V using Linux interrupt handler | 3.94 |
| Hyper-V using VMBus interrupt handler | 3.99 |
| VirtualBox | 3.96 |
| ESXi | 3.96 |
| Xen | 3.95 |

Table 3: Markov test minimum entropy of interrupt time deltas on different VMMs with step size 1

The test data for Microsoft Hyper-V (standard Linux interrupt handler as well as VMBus interrupt handler), VMWare ESXi, Oracle VirtualBox, and Xen show very similar test results.

**IRQ Noise Source Without High-Resolution Timer** The design of the LRNG supports systems without high-resolution time stamps. Testing of that noise source is not yet conducted as the results will be a challenge to interpret.

Due to the large uncertainty about the distribution, the `LRNG_IRQ_OVERSAMPLING_FACTOR` is set to the value of 10. This implies that when the hardware does not provide a high-resolution time stamp, the entropy assumed to be present with each interrupt is one 10th of the entropy with high-resolution time stamps.

### 3.4.2 IRQ Noise Source – LFSR Processed Data

Using the time stamp data stream processed with the LFSR discussed in section 3.4.1, the following analyses were conducted.

The SystemTap script `lrng_irq_raw_entropy.stp` is used to obtain the individual words of the entropy pool when processing the respective word with the LFSR. Therefore, the result using that script shows the distribution of data in the entropy pool.

The read words of the entropy pool are analyzed with `ent` showing the following results.

Listing 8: Statistical results of folded delta -- `ent` tool

```
$ ent -b irq_raw_time.out.bin
```

```
Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 275691904 bit file by 0 percent.

Chi square distribution for 275691904 samples is 0.11, and randomly
would exceed this value 73.92 percent of the times.

Arithmetic mean value of data bits is 0.5000 (0.5 = random).
Monte Carlo value for Pi is 3.141640033 (error 0.00 percent).
Serial correlation coefficient is 0.000029 (totally uncorrelated = 0.0).

$ ent irq_raw_time.out.bin
Entropy = 7.999995 bits per byte.

Optimum compression would reduce the size
of this 34461488 byte file by 0 percent.

Chi square distribution for 34461488 samples is 235.64, and randomly
would exceed this value 80.25 percent of the times.

Arithmetic mean value of data bytes is 127.5047 (127.5 = random).
Monte Carlo value for Pi is 3.141640033 (error 0.00 percent).
Serial correlation coefficient is -0.000051 (totally uncorrelated = 0.0).
```

The Chi-Square result from `ent` shows a perfect result indicating that the data stream is considered to be white noise.

The SP800-90B minimum entropy tests are conducted with the same data stream and show the results given in table 4. The calculated minimum entropy is very close to the 1 bit of entropy per data bit mark, indicating that the data set has almost full entropy.

| Test | Minimum Entropy per Data Bit |
|---|---|
| Bins Test | 0.998 |
| Collision Test | 0.96 |
| Compression Test | 0.96 |
| Collection Test | 0.98 |
| Markov Test with Step 1 | 0.998 |

Table 4: Minimum entropy tests (SP800-90B) of LFSR-processed data

The SP800-22rev1a statistical tests all pass.

Finally, the data stream was processed with `bzip2`:

Listing 9: bzip2 compression of folded data

```
$ ls -l irq_raw_time.out.bin
-rwxr-x---. 1 sm sm 34461488 20. Mai 23:32 irq_raw_time.out.bin
$ bzip2 -9 irq_raw_time.out.bin
$ ls -l irq_raw_time.out.bin.bz2
-rwxr-x---. 1 sm sm 34615128 20. Mai 23:32 irq_raw_time.out.bin.bz2
```

The file size after the "compression" operation is larger than the initial file, indicating that no compression was possible and only the `bzip2` meta data were added.

The AIS20/31 test procedure A is completed where all tests pass.

All mentioned tests were re-performed with the different LFSR polynomials found in the LRNG code. All results are equal to the aforementioned results, indicating that the LFSR polynomials are appropriate and that the size of the entropy pool does not influence the statistical properties given the chosen LFSR polynomials.

### 3.4.3 Distribution of Time Stamps During Boot

The description in chapter 2 explains that the LRNG is seeded early during the boot process. To support that claim, the following analysis has been done:

- Record the first 50 time stamps recorded for interrupt events by the LRNG.
- Reboot the system 50,000 times to obtain the initial time stamps for each of that boot.

To ensure that testing of a worst-case is conducted, all measurements were obtained with a test system executing as a KVM guest with a minimal set of devices and minimal set of software.

The test is implemented with `boottime_test_record.sh` supported by `boottime_test_record.service`. To execute the test, the following steps are performed:

1. Apply patch `boottime_test.diff` and compile LRNG code – this patch adds the logic to record the time stamps after boot.
2. Copy `boottime_test_record.sh` to `/usr/local/sbin` and make it executable. Also, execute `restorecon` if applicable.
3. Copy `boottime_test_record.service` to `/etc/systemd/system/`.
4. Execute `systemctl enable boottime_test_record`.
5. Reboot and wait until reboot test completes. The test can be interrupted by supplying the kernel command line argument of `boottime_test_stop` which boots the system in normal mode. After the interruption, the test can be continued by simply rebooting it again.
6. Execute `boottime_test_conv.pl` with the test results to convert them into time deltas.
7. Execute `boottime_test_dist.r` with result from step 6 to obtain the statistical results.

To compare the different boot time stamps, it is not appropriate to compare the absolute time stamps as they depend on the state of high-resolution time stamp. If the test is executed in a virtual machine, the high-resolution time stamp may continue to tick during the reboot operation. Therefore, the analysis must focus on the time deltas, i.e. the values that contain the variations which are believed to hold entropy. Using this idea, the histogram in figure 3.3 is generated. This figure shows the histogram of the time delta of the first and second time stamp obtained from interrupts after boot.

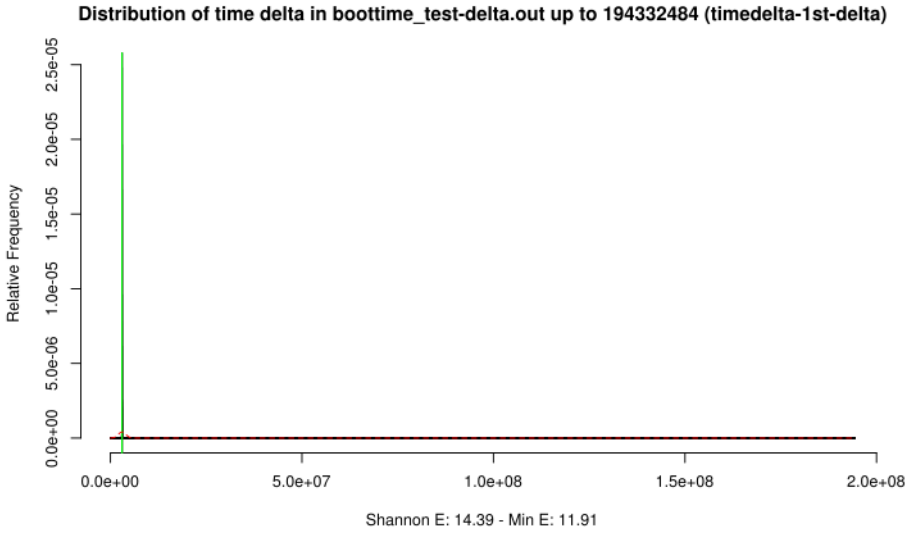


Figure 3.3: Distribution of first time delta

As figure 3.3 spans a wide spectrum of values, the distribution is not too visible. To make the distribution more visible, the 99% quartile is printed in figure 3.4 which removes outliers and allows a better view of the distribution.

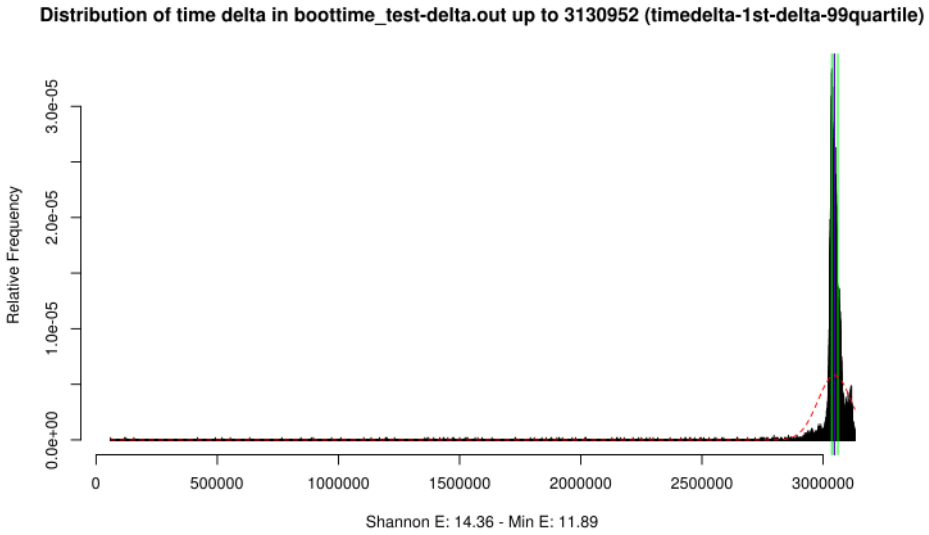


Figure 3.4: 99% Quartile of first time delta

The distribution in figure 3.4 shows large variations of the time delta, i.e. the variable that contains the entropy. Looking at the calculated Minimum Entropy and Shannon Entropy values in the legend of the figures demonstrate

that even the time delta variations of the first two time stamps after boot are similar to the variations in the time delta at runtime of the system depicted in figure 3.2. Therefore, it can be concluded that the variations of the time delta during boot indicates that entropy for the LRNG is present to an equal level as found during runtime.

To further support the analysis, the following table is generated which lists the mean, the standard derivation, Minimum Entropy and Shannon Entropy values for all 49 recorded time deltas:

| Time Delta | Mean | Sigma | Min E. | Shannon E. |
|---|---|---|---|---|
| 1 | 3062833.8 | 969143.6 | 11.9 | 14.4 |
| 2 | 3119590.1 | 5955328.8 | 10.6 | 13.0 |
| 3 | 3119542.3 | 1183534.2 | 10.4 | 12.7 |
| 4 | 3169123.3 | 6701369.9 | 10.9 | 13.1 |
| 5 | 3335203.8 | 14068575.8 | 9.5 | 12.0 |
| 6 | 3658025.1 | 22490319.7 | 9.3 | 11.9 |
| 7 | 41504201.3 | 29750931.0 | 13.3 | 15.4 |
| 8 | 31335528.7 | 83917312.0 | 13.3 | 15.4 |
| 9 | 348389141.3 | 414633897.4 | 13.6 | 15.5 |
| 10 | 1048577313.6 | 537924364.5 | 14.0 | 15.6 |
| 11 | 24755671.9 | 167820762.9 | 13.3 | 15.4 |
| 12 | 27791464.6 | 174272417.6 | 12.3 | 14.9 |
| 13 | 40291607.2 | 229038419.5 | 12.0 | 14.7 |
| 14 | 6101191.6 | 87888176.4 | 12.4 | 15.1 |
| 15 | 1871061.6 | 46665270.7 | 12.4 | 15.0 |
| 16 | 551066.8 | 18829722.1 | 12.6 | 15.0 |
| 17 | 304499.3 | 10237540.1 | 12.3 | 14.9 |
| 18 | 256659.1 | 5857185.4 | 12.1 | 14.9 |
| 19 | 304808.6 | 1953846.1 | 12.3 | 15.0 |
| 20 | 627977.5 | 10294735.3 | 13.3 | 15.4 |
| 21 | 176912.7 | 1646102.2 | 11.7 | 14.4 |
| 22 | 442046.3 | 9978753.7 | 12.6 | 15.2 |
| 23 | 158246.7 | 963124.3 | 12.0 | 14.7 |
| 24 | 222521.6 | 1258827.4 | 12.3 | 14.8 |
| 25 | 214526.3 | 1151931.8 | 12.1 | 14.8 |
| 26 | 154615.4 | 910690.8 | 11.1 | 14.4 |
| 27 | 177672.6 | 1004500.7 | 11.7 | 14.5 |
| 28 | 189688.2 | 1078282.8 | 11.8 | 14.5 |
| 29 | 165000.7 | 923889.6 | 11.2 | 14.1 |
| 30 | 146079.5 | 878573.1 | 11.0 | 14.0 |
| 31 | 153658.9 | 792815.2 | 11.0 | 14.1 |
| 32 | 162527.8 | 881274.2 | 11.3 | 14.3 |
| 33 | 155808.9 | 816388.3 | 11.0 | 14.0 |
| 34 | 141656.0 | 769105.7 | 10.9 | 13.9 |
| 35 | 154833.1 | 825433.4 | 10.9 | 14.0 |
| 36 | 161944.1 | 807917.6 | 11.2 | 14.1 |
| 37 | 165380.7 | 824345.6 | 11.0 | 14.2 |
| 38 | 149306.4 | 683709.3 | 11.4 | 14.3 |

| | | | | |
|---|---|---|---|---|
| 39 | 162143.9 | 791188.8 | 11.1 | 14.4 |
| 40 | 180302.5 | 874298.3 | 11.8 | 14.7 |
| 41 | 176926.1 | 925924.0 | 11.9 | 14.6 |
| 42 | 141931.2 | 594531.7 | 11.4 | 14.4 |
| 43 | 156627.0 | 760098.0 | 10.9 | 14.3 |
| 44 | 164832.6 | 822648.6 | 11.5 | 14.4 |
| 45 | 151223.6 | 692047.8 | 11.5 | 14.4 |
| 46 | 131969.9 | 594910.4 | 11.5 | 14.3 |
| 47 | 141684.6 | 644875.9 | 11.6 | 14.5 |
| 48 | 113142.1 | 669510.4 | 11.4 | 14.2 |
| 49 | 151697.0 | 630606.8 | 11.5 | 14.6 |

The numbers presented in the table support the conclusion drawn for the first time delta: during boot time, sufficient variations of the time deltas are present to support the LRNG's entropy estimate of one bit of entropy per interrupt event. The variations are equal in size compared to the runtime of the system.

### 3.4.4 DRNG Output – /dev/random

The test of the output of /dev/random is performed on post-whitened data. Hence, statistical tests are not meaningful any more to demonstrate the entropy content. The test only demonstrates a proper implementation of the output function avoiding implementation errors like CVE:2013-4345.

The DRNG is read via /dev/random using a prime block size showing that the read/write pointer offset handling is done correctly. To fill the DRNG with entropy, a large number of interrupts are generated using the ping flood again.

Listing 10: Statistical properties of LRNG /dev/random output with prime block size

```
$ dd if=/dev/random of=file bs=29
^C32179+32179 Datensätze ein
32179+32179 Datensätze aus
1029728 bytes (1,0 MB, 1006 KiB) copied, 486,352 s, 2,1 kB/s

--- 130 sm@x86-64 ~ ------------------------------------------------------------
$ ent -b file
Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 8237824 bit file by 0 percent.

Chi square distribution for 8237824 samples is 0.06, and randomly
would exceed this value 80.14 percent of the times.

Arithmetic mean value of data bits is 0.5000 (0.5 = random).
Monte Carlo value for Pi is 3.140944290 (error 0.02 percent).
Serial correlation coefficient is 0.000093 (totally uncorrelated = 0.0).
--- 0 sm@x86-64 ~ -------------------------------------------------------------
$ ent file
Entropy = 7.999809 bits per byte.

Optimum compression would reduce the size
of this 1029728 byte file by 0 percent.

Chi square distribution for 1029728 samples is 272.84, and randomly
would exceed this value 21.14 percent of the times.

Arithmetic mean value of data bytes is 127.4439 (127.5 = random).
Monte Carlo value for Pi is 3.140944290 (error 0.02 percent).
```

```
Serial correlation coefficient is 0.001174 (totally uncorrelated = 0.0).
```

Since the Chi-Square test result indicates white noise, a proper implementation of the read/write offset pointer handling is concluded.

This test has been performed for all three types of DRNG defined in the LRNG source code. The test results are all equivalent.

### 3.4.5 DRNG Output – /dev/urandom

Similarly to the /dev/random testing, /dev/urandom is read to check for potential implementation errors:

Listing 11: Statistical properties of LRNG /dev/urandom output

```
$ dd if=/dev/urandom of=file count=10000
10000+0 Datensätze ein
10000+0 Datensätze aus
5120000 bytes (5,1 MB, 4,9 MiB) copied, 0,906385 s, 5,6 MB/s
--- 0 sm@x86-64 ~ ------------------------------------------------------------
$ ent file
Entropy = 7.999964-bits per byte.

Optimum compression would reduce the size
of this 5120000 byte file by 0 percent.

Chi square distribution for 5120000 samples is 256.75, and randomly
would exceed this value 45.74 percent of the times.

Arithmetic mean value of data bytes is 127.4657 (127.5 = random).
Monte Carlo value for Pi is 3.142787165 (error 0.04 percent).
Serial correlation coefficient is -0.000281 (totally uncorrelated = 0.0).
--- 0 sm@x86-64 ~ ------------------------------------------------------------
$ ent -b file
Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 40960000 bit file by 0 percent.

Chi square distribution for 40960000 samples is 0.66, and randomly
would exceed this value 41.58 percent of the times.

Arithmetic mean value of data bits is 0.5001 (0.5 = random).
Monte Carlo value for Pi is 3.142787165 (error 0.04 percent).
Serial correlation coefficient is 0.000098 (totally uncorrelated = 0.0).
```

The `ent` tool's Chi-Square indicates white noise and thus does not give any hints to implementation errors.

Note, this type of output was obtained for all different types of DRNG showing a perfect Chi-Square test result.

### 3.4.6 /dev/urandom Performance And DRNG Type

As documented above, the LRNG is capable of using all types of DRNG provided by the Linux kernel. On the test system that executes within a KVM and on top of an Intel Core i7 Broadwell CPU[19], the following read speeds using the command `dd if=/dev/urandom of=file count=50000 bs=XX` are obtained where `bs` is set to the read size values indicated in the following tables. These numbers give an indication on how much one DRNG performs better over another[20]

[19] This CPU offers AES-NI, and AVX2 that is used by the allocated AES and SHA implementations.

[20] Please note that the test system is a 64-bit system. On 64-bit systems, SHA-512 is faster by a factor of almost 2 compared to SHA-256 when the output data size is segmented into 64 bytes – the SHA-512 block size.

and are presented in table 6. This table lists the DRNG type, the type and implementation of the underlying cipher and the performance in MBytes per second. Please note that the read sizes have been chosen as follows: The small read sizes are based on the buffer size of the used DRNG and do not require a `kmalloc` call in the `lrng_read_common` function. The other values shall indicate the performance when using higher block sizes up to the point the maximum request size is reached. The read size of the legacy /dev/random is hard coded to 10 bytes by the kernel.

| DRNG Type | Cipher | Cipher Impl. | Read Size | Performance |
|---|---|---|---|---|
| HMAC DRBG | SHA-512 | C | 64 bytes | 13.8 MB/s |
| HMAC DRBG | SHA-512 | AVX2 | 16 bytes | 4.1 MB/s |
| HMAC DRBG | SHA-512 | AVX2 | 32 bytes | 8.1 MB/s |
| HMAC DRBG | SHA-512 | AVX2 | 64 bytes | 15.7 MB/s |
| HMAC DRBG | SHA-512 | AVX2 | 128 bytes | 26.1 MB/s |
| HMAC DRBG | SHA-512 | AVX2 | 4096 bytes | 69.3 MB/s |
| Hash DRBG | SHA-512 | C | 64 bytes | 27.9 MB/s |
| Hash DRBG | SHA-512 | AVX2 | 16 bytes | 9.4 MB/s |
| Hash DRBG | SHA-512 | AVX2 | 32 bytes | 18.8 MB/s |
| Hash DRBG | SHA-512 | AVX2 | 64 bytes | 36.5 MB/s |
| Hash DRBG | SHA-512 | AVX2 | 128 bytes | 57.5 MB/s |
| Hash DRBG | SHA-512 | AVX2 | 4096 bytes | 145 MB/s |
| CTR DRBG | AES-256 | C | 16 bytes | 15.4 MB/s |
| CTR DRBG | AES-256 | AES-NI | 16 bytes | 17.1 MB/s |
| CTR DRBG | AES-256 | AES-NI | 32 bytes | 34.3 MB/s |
| CTR DRBG | AES-256 | AES-NI | 64 bytes | 67.9 MB/s |
| CTR DRBG | AES-256 | AES-NI | 128 bytes | 128.0 MB/s |
| CTR DRBG | AES-256 | AES-NI | 4096 bytes | 941.3 MB/s |
| ChaCha20 | ChaCha20 | C | 16 bytes | 32.8 MB/s |
| ChaCha20 | ChaCha20 | C | 32 bytes | 65.0 MB/s |
| ChaCha20 | ChaCha20 | C | 64 bytes | 103.8 MB/s |
| ChaCha20 | ChaCha20 | C | 128 bytes | 162.0 MB/s |
| ChaCha20 | ChaCha20 | C | 4096 bytes | 472.6 MB/s |
| Legacy /dev/random | SHA-1 | C | 10 bytes | 12.9 MB/s |
| Legacy /dev/random | ChaCha20 | C | 16 bytes | 29.2 MB/s |
| Legacy /dev/random | ChaCha20 | C | 32 bytes | 58.6 MB/s |
| Legacy /dev/random | ChaCha20 | C | 64 bytes | 80.0 MB/s |
| Legacy /dev/random | ChaCha20 | C | 128 bytes | 118.7 MB/s |
| Legacy /dev/random | ChaCha20 | C | 4096 bytes | 220.2 MB/s |

Table 6: LRNG /dev/urandom performance on 64-bit

In addition, table 7 documents the performance on 32 bit using the same hardware to have a comparison to the 64-bit case. Note, the CTR DRBG performance for large blocks can be increased to more than 2 GB/s when `DRBG_CTR_NULL_LEN` and `DRBG_OUTSCRATCHLEN` in `crypto/drbg.c` is increased to 4096.

| DRNG Type | Cipher | Cipher Impl. | Read Size | Performance |
|---|---|---|---|---|
| HMAC DRBG | SHA-512 | C | 16 bytes | 1.2 MB/s |
| HMAC DRBG | SHA-512 | C | 32 bytes | 2.3 MB/s |
| HMAC DRBG | SHA-512 | C | 64 bytes | 4.5 MB/s |
| HMAC DRBG | SHA-512 | C | 128 bytes | 7.4 MB/s |
| HMAC DRBG | SHA-512 | C | 4096 bytes | 16.5 MB/s |
| Hash DRBG | SHA-512 | C | 16 bytes | 2.9 MB/s |
| Hash DRBG | SHA-512 | C | 32 bytes | 5.6 MB/s |
| Hash DRBG | SHA-512 | C | 64 bytes | 10.7 MB/s |
| Hash DRBG | SHA-512 | C | 128 bytes | 17.4 MB/s |
| Hash DRBG | SHA-512 | C | 4096 bytes | 38.3 MB/s |
| CTR DRBG | AES-256 | AES-NI | 16 bytes | 12.1 MB/s |
| CTR DRBG | AES-256 | AES-NI | 32 bytes | 22.6 MB/s |
| CTR DRBG | AES-256 | AES-NI | 64 bytes | 39.0 MB/s |
| CTR DRBG | AES-256 | AES-NI | 128 bytes | 60.7 MB/s |
| CTR DRBG | AES-256 | AES-NI | 4096 bytes | 146.0 MB/s |
| ChaCha20 | ChaCha20 | C | 16 bytes | 24.9 MB/s |
| ChaCha20 | ChaCha20 | C | 32 bytes | 50.2 MB/s |
| ChaCha20 | ChaCha20 | C | 64 bytes | 75.5 MB/s |
| ChaCha20 | ChaCha20 | C | 128 bytes | 119.2 MB/s |
| ChaCha20 | ChaCha20 | C | 4096 bytes | 308.1 MB/s |
| Legacy /dev/random | SHA-1 | C | 10 bytes | 9.4 MB/s |
| Legacy /dev/random | ChaCha20 | C | 16 bytes | 21.0 MB/s |
| Legacy /dev/random | ChaCha20 | C | 32 bytes | 43.0 MB/s |
| Legacy /dev/random | ChaCha20 | C | 64 bytes | 59.4 MB/s |
| Legacy /dev/random | ChaCha20 | C | 128 bytes | 88.6 MB/s |
| Legacy /dev/random | ChaCha20 | C | 4096 bytes | 166.7 MB/s |

Table 7: LRNG /dev/urandom performance on 32 bit

Note, to enable the different cipher implementations, they need to be statically linked into the kernel binary.

To ensure that the respective implementations of the cipher cores are used, they must be statically linked into the kernel.

The reason for the fast processing of larger read requests lies in the concept of the DRBG: the DRBG generates the requested number of bytes followed by an update operation which generates a new internal state. Thus, the larger the generate requests are, the less number of state update operations are performed relative to the data size. The LRNG enforces that at most $2^{12}$ bytes are generated before an update is enforced as documented in section 2.8.1.

### 3.4.7 ChaCha20 Random Number Generator

The ChaCha20 DRNG is analyzed to verify the following properties:

- whether the self-feeding RNG ensures backtracking resistance, and
- whether the absence of the CPU noise source still produces white noise.

The compilation of the LRNG code is changed such that the ChaCha20 DRNG is compiled. Also, for testing, the fast noise sources have been disabled to clearly

demonstrate that the backtracking resistance is ensured. This is followed by obtaining random numbers from /dev/urandom and calculating the statistical properties:

Listing 12: Statistical properties of ChaCha20 RNG with interrupt noise source

```
--- 0 sm@x86-64 ~ ---------------------------------------------------------------
$ dd if=/dev/urandom of=file count=1000
1000+0 Datensätze ein
1000+0 Datensätze aus
512000 bytes (512 kB, 500 KiB) copied, 0,00341658 s, 150 MB/s
--- 0 sm@x86-64 ~ ---------------------------------------------------------------
$ ent file
Entropy = 7.999639 bits per byte.

Optimum compression would reduce the size
of this 512000 byte file by 0 percent.

Chi square distribution for 512000 samples is 257.07, and randomly
would exceed this value 45.19 percent of the times.

Arithmetic mean value of data bytes is 127.4761 (127.5 = random).
Monte Carlo value for Pi is 3.147902921 (error 0.20 percent).
Serial correlation coefficient is 0.001163 (totally uncorrelated = 0.0).
--- 0 sm@x86-64 ~ ---------------------------------------------------------------
$ ent -b file
Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 4096000 bit file by 0 percent.

Chi square distribution for 4096000 samples is 0.12, and randomly
would exceed this value 73.24 percent of the times.

Arithmetic mean value of data bits is 0.5001 (0.5 = random).
Monte Carlo value for Pi is 3.147902921 (error 0.20 percent).
Serial correlation coefficient is 0.000028 (totally uncorrelated = 0.0).
```

The Chi-Square result indicates white noise and thus allows the conclusion that the ChaCha20 DRNG operates as expected and that backtracking resistance is implemented correctly.

A fully stand-alone user-space implementation of the ChaCha20 DRNG is provided at the ChaCha20 DRNG website. This implementation is an extraction of the ChaCha20-based DRNG used for the LRNG and is provided to allow studying the ChaCha20-based DRNG without the limitation of kernel space.

## 3.5 Usability Tests

To show that the LRNG is a drop-in replacement for the legacy /dev/random, the following usability tests were executed:

- Parallel execution of `cat /dev/urandom > /dev/null` on all available cores for half a day to verify stability. This test allowed verification of the reseed operation of the secondary DRNG when reaching the threshold for the maximum number of random numbers to be generated.
- Parallel execution of `cat /dev/urandom > /dev/urandom` on all available cores for half a day to verify stability.
- Parallel execution of `cat /dev/random > /dev/null` on all available cores for an hour to verify stability.
- Execution of `cat /dev/urandom > /dev/urandom` and verification that writing of data from user space into the device files does not reset the

reseed threshold or the reseed timer for the secondary DRNG. I.e. even while executing this command, the secondary DRNG reseeds after 600 seconds or reaching $2^{20}$ requests, whatever is reached first. This behavior ensures that unprivileged user space cannot block the reseeding of the secondary DRNG.

- Execution of `cat /dev/random` without fast noise sources to drain the entropy pool and then send a ping flood to the test system to verify that /dev/random resumes generation of random data when entropy is received via new interrupts.

- Execution of `cat /dev/random` without fast noise sources to drain the entropy pool and then observe /proc/sys/kernel/random/entropy_avail to reach the value in /proc/sys/kernel/random/read_wakeup_threshold. When the threshold is reached, new data is printed with the cat command. This test is repeated by setting the read_wakeup_threshold to the minimum value of 32 to verify that new data is printed with the cat command when reaching this lower value. This test verifies that the read_wakeup_threshold is enforced properly and that the reader wakeup calls are placed at the right spots in the LRNG code.

- During the parallel execution of `cat /dev/random` without fast noise sources the entropy pool is carefully filled until reaching the wakeup threshold. Upon reaching the wakeup threshold, it is verified that only one cat process is woken up and returns random data. This test shows that entropy is only used once.

- Use of an the `jitterentropy-rngd`[21] user space daemon, disabling the fast noise sources and execution of `cat /dev/random` to verify that the injection of entropy via the `RNDADDENTROPY` IOCTL resumes the generation of random data for /dev/random. In addition, this test shows that the user space daemon as well as the LRNG is woken up at the right time – i.e. the reader and writer wakeup calls in the LRNG are placed at the right spots.

- Leaving the LRNG in peace on a very quiet system to verify that the automatic reseed operation after reaching the timer-based reseed threshold is performed. This test also shows that the entropy gathered within that time frame from interrupts with disabled fast noise sources is more than 256 bits. I.e. the reseeding after the expiry of the timer on a very quiet system will not drain the entropy pool.

- Executing `cat /dev/urandom > /dev/random` and in parallel executing `cat /dev/random` where the LRNG does not use the fast noise sources. It is expected that /dev/random will not produce random data beyond the point where the entropy pool is drained. This test shall show that a simple writing of data into /dev/random or /dev/urandom will not be considered as entropic data to generate fresh data for /dev/random.

- Execution of the LRNG in a virtual environment to verify that it receives sufficient entropy there as well.

[21] This daemon is available at http://www.chronox.de/jent.html.

- Execution of the LRNG on x86 32-bit, x86 64-bit, ARM 32-bit, and MIPS 32-bit systems.
- Execution of a stress test to switch the DRNG implementations while /dev/random and /dev/urandom must generate large amounts of data. This test is provided in the implementation `swap_stress.sh`.

# 4 Integration Into Linux Kernel

The LRNG can be integrated into the kernel in different ways. The following options are currently seen:

1. Replacing the legacy /dev/random implementation in `drivers/char/random.c` which implies that the kernel crypto API along with its DRNG must always be compiled.
2. Make the LRNG a compile time option that can be enabled when the kernel crypto API is selected. If the LRNG is selected, the legacy /dev/random code is not compiled. This would allow users not wanting the kernel crypto API always in the kernel to keep it excluded.
3. Having the LRNG changed to use the SHA-1 implementation from lib/sha1.c and implement its own DRNG. This has the drawback that the LRNG is stuck with a software SHA-1 implementation. Also, there would potentially two implementations of the DRNG be present in the kernel. Finally, the private DRNG implementation may be hard to test and validate – the use of a testable and validated DRNG as one of the goals of the chosen LRNG approach would be lost.
4. Use the current DRNG implementations and replace the cipher invocations to call the `lib/sha1.c` code. This would involve extensive changes to `crypto/drbg.c`. Furthermore, this would limit the choice to a Hash DRBG with SHA-1 core and a cryptographic strength of 128 bits as defined in SP800-90A.

The author would disregard options 3 and 4. Options 1 and 2 are considered technically equal.

To support a first testing phase of the LRNG, a patch following option 2 is provided.

# A Thanks

Special thanks for providing input as well as mathematical support goes to:

- DJ Johnston
- Yi Mao
- Sandy Harris
- Dr. Matthias Peter
- Quentin Gouchet

# B Source Code Availability

The source code, this document as well as the test code for all aforementioned tests is available at http://www.chronox.de/lrng.html.

# C Bibliographic Reference

# References


[1] Elaine Barker and John Kelsey. *NIST Special Publication 800-90A Recommendation for Random Number Generation using Deterministic Random Bit Generators*. 2012.

[2] Elaine Barker and John Kelsey. *NIST DRAFT Special Publication 800-90C Recommendation for Random Bit Generatior (RBG) Constructions*. 2012.

[3] Elaine Barker and John Kelsey. *NIST DRAFT Special Publication 800-90B Recommendation for the Entropy Sources Uses for Random Bit Generation*. 2nd draft edition, 2016.

[4] Elaine Barker and John Kelsey. *(Second Draft) Special Publication 800-90C Recommendation for Random Bit Generatior (RBG) Constructions*. 2016.

[5] Elaine Barker and Allen Roginsky. *NIST DRAFT Special Publication 800-131A Revision 1 Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths*. 2015.

[6] BSI. *BSI - Technische Richtline TR-02102-1*. 2016.

[7] Wolfgang Killmann and Werner Schindler. *AIS 20/31: A proposal for: Functionality classes for random number generators*. 2011.

[8] Stephan Müller. *Dokumentation und Analyse des Linux-Pseudozufallszahlengenerators*. 2013.

[9] Stephan Müller. /dev/random and sp800-90b. International Cyptographic Module Conference (ICMC), 2015.

[10] Stephan Müller. *Analysis of Random Number Generation in Virtual Environments*. 2016.

[11] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539 (Informational), May 2015. URL http://www.ietf.org/rfc/rfc7539.txt.

[12] Andrew Rukhin, Juan Soto, James Nechvatal, Elaine Barker Miles Smid, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Revision 1a edition, 2010.

# D License

The implementation of the Linux Random Number Generator, all support mechanisms, the test cases and the documentation are subject to the following license.

Copyright Stephan Müller <smueller@chronox.de>, 2016.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, and the entire permission notice in its entirety, including the disclaimer of warranties.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

ALTERNATIVELY, this product may be distributed under the terms of the GNU General Public License, in which case the provisions of the GPL are required INSTEAD OF the above restrictions. (This clause is necessary due to a potential bad interaction between the GPL and the restrictions contained in a BSD-style copyright.)

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ALL OF WHICH ARE HEREBY DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF NOT ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.